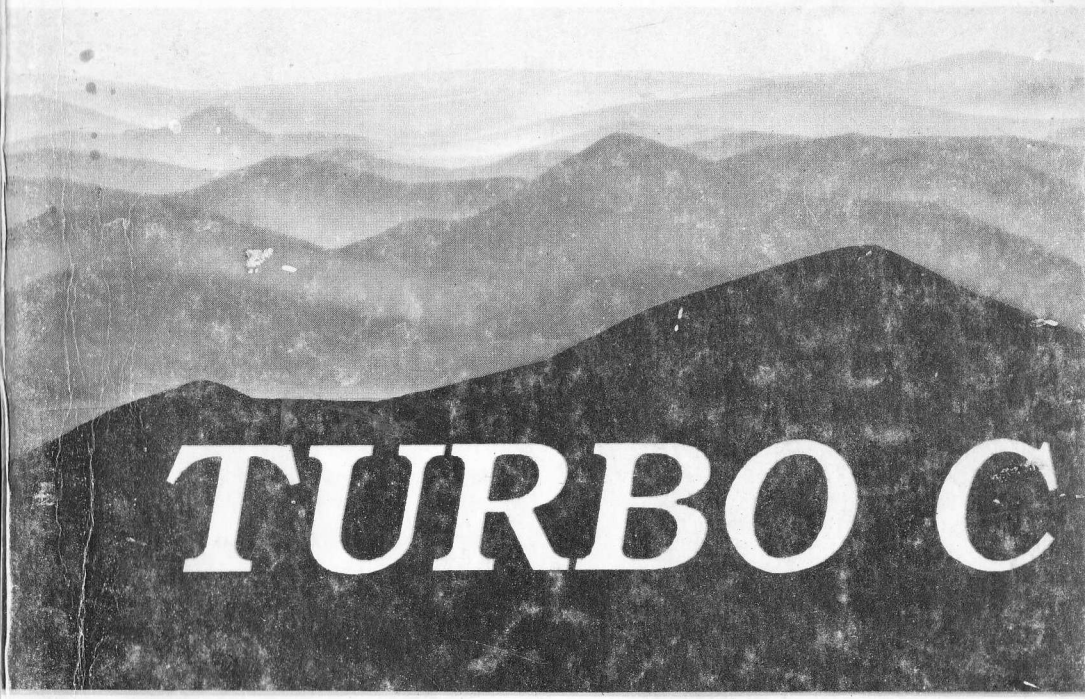
BIBLIOTECA
PC



TURBO C

Tehnici de programare

micro informatica srl
EDITURA MICROINFORMATICA

Str. Observatorului nr. 1, bl. OS1,
Cluj-Napoca, 3400
Oficiul P.T.T.R. Cluj-Napoca 1, C.P. 186
tel. 95/118263

Cartea de față este dedicată tuturor acelora care încearcă sau au reușit deja să patrundă în tainele programării în limbajul C. Ea este axată pe prezentarea detaliată a unor tehnici de programare rar sau deloc întîlnite în literatura de specialitate sau care nu există în alte limbaje de programare de nivel înalt.
Prin ea se poate dobîndi abilitatea scrierii unui cod C clar și eficient indiferent de experiența în programare. In acest scop sînt deosebit de utile sutele de linii de cod sursă C cuprinse în cartea de față.

ISBN 973-95718-1-6

Lei 425,–

Cuprins

	Prefață	5
1.	Scurte considerații teoretice	7
1.1.	Operatori	7
1.2.	Instrucțiuni de control a fluenței programului	15
1.3.	Șiruri și pointeri	22
1.4.	Argumentele liniei de comandă	26
1.5.	Funcții	29
1.6.	Structuri și uniuni	46
1.7.	Clase de memorie	49
1.8.	Directive preprocesor și macrouri	55
1.9.	Castul	59
1.10.	Probleme propuse pentru rezolvare	59
2.	Mediul de execuție	65
2.1.	Zone program	65
2.2.	Zona text	66
2.3.	Zona stivă	70
2.4.	Zona de date	84
2.5.	Zona Heap	96
2.6.	Probleme propuse pentru rezolvare	108
3.	Alternative și posibilități	111
3.1.	Șiruri și pointeri	111
3.2.	Expresii cu pointeri și șiruri	116
3.3.	Șiruri multidimensionale	133
3.4.	Expresii compacte cu pointeri în comparație cu șiruri indexate	146
3.5.	Șiruri multidimensionale în momentul execuției	166
3.6.	Probleme	187
4.	O privire generală asupra limbajului C	193
4.1.	Secvențe garantate	193
4.2.	Regula dreapta-stînga	200
4.3.	L-valori în expresii	209
4.4.	Transmiterea șirurilor	213

4.5. Transmiterea şirurilor ca parametri pentru funcţii 217
4.6. Funcţii cu număr variabil de argumente 222
4.7. Probleme 229

5. Tehnici de depanare 232
5.1. Preprocesorul C 232
5.2. Teste de interfaţa înglobate 236
5.3. Mesaje de informare în format ASCII şi hexa 239
5.4. Fanioanele liniei de comandă 243
5.5. Semnale şi informaţii de depanare 250
5.6. Aserţiuni 253
5.7. Afişarea selectivă a informaţiilor de depanare 268
5.8. Observaţii asupra funcţiilor de gestionare a memoriei 273
5.9. Probleme 281

6. Alocator de memorie orientat pe obiect 282
6.1. Consideraţii teoretice 282
6.1.1. Colectarea reziduurilor 283
6.2. Alocatorul de obiecte 283
6.2.1. Obiecte şi variabile de control 284
6.2.2. Rutinele alocatorului 286
6.2.3. Redimensionarea obiectelor 294
6.2.4. Compactarea datelor obiect 297
6.2.5. Verificarea consistenţei 299
6.2.6. Interfaţa alocatorului 302
6.2.7. Depanarea alocatorului 305

A1. Anexă - Funcţii video Turbo C 2.0 307
A1.1. Modul text 307
A1.2. Modul grafic 313

Bibliografie 336

Prefaţă

Limbajul C se foloseşte de 5-20 de ani şi este astăzi mai popular ca oricînd. Fiind considerat la începutul anilor '70 un limbaj original de programare pentru minicalculatoare, a devenit în anii '80 un standard în programare, un limbaj care prezintă facilităţi şi tehnici deosebite. Acest lucru a devenit posibil datorită industriei microprocesoarelor, performanţele software fiind indisolubil legate de noile tehnologii din hardware. Limbajul C a luat un avans faţă de celelalte limbaje de programare, datorită capacităţii sale de a realiza operaţii la nivel de maşină, precum şi portabilităţii sale.

Desigur, cînd un limbaj s-a extins atît de mult ca şi C-ul, apare necesitatea prezentării unor tehnici. Cartea **Tehnici de programare în limbajul C** se adresează programatorilor interesaţi în aprofundarea acestui limbaj, pentru o utilizare a sa cît mai eficientă. Tehnicile prezentate în această lucrare sînt rezultatul elaborării unui număr mare de programe C şi a unei intense activităţi de informare în domeniu.

Unul dintre obiectivele majore ale lucrării de faţă îl constituie prezentarea unor tehnici avansate, des întîlnite de programatorii C experimentaţi, dar care au o foarte mică sau chiar nici o documentaţie. Modul de abordare al limbajului C este, de aceea, puţin diferit de cel al altor cărţi din domeniu. În loc de a vă prezenta cum să transpuneţi tehnici din alte limbaje de programare în C (cum ar fi liste înlănţuite, arbori binari, şamd), lucrarea este axată pe dezvoltarea unor tehnici ale limbajului, care nu există în alte limbaje de programare. Şi, cu toate că aceasta face ca tehnicile respective să nu fie aplicabile în alte limbaje, nu înseamnă că ele sînt mai puţin valoroase.

S-a presupus că sintaxa limbajului este cunoascută şi că au fost trecute în revistă funcţiile sale de bibliotecă. Programele prezentate în lucrare pot fi testate pe calculatoare cu sistem de operare MS-DOS, sau OS/2.

În realizarea programelor din volumul de faţă am făcut eforturi ca acestea să fie cît mai portabile (în măsura posibilităţilor). Ele sînt elaborate pe calculatoare care includ procesoare 8086, 80186, 80286 şi 80386. Am ales această arhitectură datorită proliferării PC-urilor în toate domeniile şi la toate nivelurile.

Dacă calculatorul dumneavoastră nu face parte din lista de mai sus, pot apărea detalii de implementare care să difere, dar principiile de lucru rămîn valabile, iar programele necesită eventual, modificări minore.

Lucrarea este formată din şase capitole. Fiecare capitol este de-sine-stătător, astfel încît programatorii C să poată studia numai capitolele care îi interesează.

La sfîrșitul fiecărui capitol se află un set de probleme propuse spre rezolvare, care îmbogățește materialul prezentat.

Capitolul 1 reprezintă o trecere în revistă a limbajului C. Evident, acesta nu este un manual de inițiere în limbajul C, pentru aceasta existînd manuale de prezentare. Capitolul 1 furnizează , însă, bazele teoretice necesare pentru tehnicile dezvoltate. Programatorilor care vor dori să treacă mai repede peste această parte, le recomandăm să arunce o privire asupra programelor incluse, care utilizează tehnici avansate.

Capitolul 2 prezintă mediul de execuție sau ceea ce se întîmplă în timpul execuției unui program C. Cunoștințele despre zonele text, date, stivă și heap vă vor ajuta să determinați cîtă memorie necesită un program, astfel încît să utilizați cît mai eficient memoria.

Capitolul 3 face legătura între conceptele de șiruri și pointeri. Aici veți descoperi tehnici care vă vor ajuta să interpretați expresiile cu pointeri, să declarați șiruri dinamice multidimensionale și să măriți eficența programelor prin utilizarea pointerilor.

Capitolul 4 tratează cîteva probleme de detaliu ale limbajului C, cum ar fi: secvențe garantate de pointeri și regula dreapta-stînga, transferul șirurilor, transmiterea unui șir întreg ca parametru la o funcție sau apelul unei funcții cu număr variabil de argumente.

Capitolul 5 ilustrează posibilitățile de utilizare a limbajului C în depanare. Metode prezentate: compilare condiționată, fanioanele liniei de comandă, aserțiuni, afișarea selectivă a informațiilor de depanare sînt independente de un depanator și vă permit să eliminați erorile mai greu vizibile.

Capitolul 6 conține codul sursă pentru un alocator de memorie pe obiecte. Programele C pot utiliza acest alocator pentru a aloca dinamic, redimensiona sau elibera obiecte din memoria heap.

Anexa de la sfîrșitul cărții prezintă funcțiile video (mod text și mod grafic) disponibile în versiunea Turbo C 2.0.

Cluj-Napoca,
Iunie 1992

Autorii

1. Scurte considerații teoretice

1.1. Operatori

Turbo C dispune de un set bogat de operatori aparținînd următoarelor clase: aritmetici, relaționali și logici, pe biți, pe pointer, de atribuire și micști.

1.1.1. Operatori aritmetici

Operator	Semnificatie
-	scadere, minus unar
+	adunare
*	inmultire
/	impartire
%	modulo
--	decrementare
++	incrementare

Tabelul 1.1.

Dacă expresia aritmetică conține doi operanzi de tipuri diferite, înainte de efectuarea operației TC convertește operandul cu tip de nivel mai scăzut la tipul operandului de nivel mai înalt. Deci rezultatul este de tipul operandului cu nivel mai înalt. Astfel:

- tipul *char* și *short* se convertesc la tipul *int*,
- tipul *float* se convertește la tipul *double*,
- dacă un operand este *long*, *unsigned* sau *double* celălalt operand se convertește respectiv la *long*, *unsigned*, *double*.
- aritmetica flotantă se face în dublă precizie.

Compilatorul generează cod în limbaj de asamblare pentru aritmetica întreagă sau flotantă funcție de regulile de mai sus.

Pentru precedența operatorilor se va consulta tabelul 1.5. Adunarea și scăderea au precedența mai mică decît ceilalți operatori. Pentru expresii ce conțin operatori de aceeași precedență tabelul conține și ordinea de evaluare. De

exemplu, operatorii aritmetici se evaluează de la stînga la dreapta astfel încît expresia:

```
40 / 4 * 5
```

se evaluează la *50* şi nu la *2*.

Operatorii aritmetici se pot aplica şi datelor de tip *char*. De exemplu, dacă variabila *c* conţine codul ASCII al unui caracter din intervalul [0, 9], instrucţiunea:

```
digit = c - '0';
```

converteşte caracterul la un număr zecimal, fără a fi nevoie de o funcţie de conversie.

1.1.2. Operatori relaţionali şi logici

Operatori relationali	Semnif atie
>	mai mare
>=	mai mare sau egal
<	mai mic
>=	mai mic sau egal
==	egal
!=	diferit

Tabelul 1.2.

Operatorii relaţionali prezentaţi sînt operatori binari si compară primul operand cu al doilea pentru a testa validitatea operaţiei specificate, producînd rezultat *1*, dacă expresia este adevărată (*TRUE*), sau *0* în caz contrar (*FALSE*). Deci rezultatul este de tip *int*.

Precedenţa şi ordinea de evaluare rezultă din tabelul 1.5.

De notat că operatorii relaţionali ce compară pointer la caracter cu şiruri constante nu produc rezultate corecte. De exemplu, declaraţia cu iniţializare:

```
char *p = "exemplu";
```


atribuie pointerului *p* adresa şirului "*exemplu*". Expresia:

```
p == "exemplu" /* comparaţie între adrese */
```

ne-am aştepta să compare pointerul *p* cu adresa şirului. Expresia nu se va evalua niciodată la *TRUE*, deoarece se alocă memorie pentru o nouă copie a şirului "*exemplu*".

Biblioteca de funcţii TC conţine funcţia *strcmp*, care permite compararea şirurilor de caractere. Funcţia *strcmp* returnează valoarea *0* dacă cele două şiruri sînt identice. Expresia:

```
strcmp( p, "exemplu") == 0
```

este TRUE dacă *p* referă şirul "*exemplu*".

Operatori logici	Semnificatie
&&	SI logic
||	SAU logic
!	negatie

Tabelul 1.3

Rezultatul unei operaţii logice este *0* sau *1*, iar tipul rezultatului este *int*.
Evaluarea pentru ŞI şi SAU este de la stînga la dreapta. Dacă primul operand evaluat este determinant pentru rezultat (adică valoarea sa este *0* iar operatorul este ŞI, sau valoarea sa este *1* şi operatorul este SAU), atunci celălalt operand nu se mai evaluează. Operatorii logici nu efectuează conversiile aritmetice.

De exemplu expresia:

```
c >= '0' && c <= '9' || c >= 'a' && c <= 'f' || c >='A' && c <='F'
```

se evaluează la *TRUE* dacă variabila *c* conţine codul ASCII al unui caracter hexazecimal.

1.1.3. Operatori pe biți

Operator	Semnificatie
&	SI pe biti
|	SAU pe biti
^	SAU exclusiv pe biti
~	Complement fata de 1
>>	deplasare dreapta
<<	deplasare stinga

Tabelul 1.4.

Operatorii pe biți oferă posibilitatea lucrului la nivel de bit. Cu excepția operatorului ~ (tilda), care schimbă orice bit *1* în *0* și invers, fiecare operator necesită doi operanzi. Operanzii pot fi de tipul *int* sau *char*. Tipul rezultatului este tipul operanzilor după efectuarea eventualelor conversii aritmetice. În cazul operatorilor >> și << , tipul rezultatului este tipul operandului stîng.

La aplicarea operatorului <<, biții din dreapta devin *0*.

La aplicarea operatorului >> biții din stînga sînt completați după tipul primului operand. Astfel, dacă tipul este *unsigned*, ei sînt completați cu *0* (deplasare logică), în caz contrar ei sînt completați cu valoarea bitului de semn (deplasare aritmetică). Rezultatul este nedefinit dacă al doilea operand este negativ.

Se observă că TC nu posedă un operator de rotire a biților. Scrierea unei funcții, care să realizeze acest lucru nu prezintă nici o dificultate. Programul **rot.c** utilizează operatorii pe bit pentru a roti un număr hexazecimal citit de la tastatură. Codul programului este:

Programul 1.1.

```
#include <stdio.h>
#define CUVDIM sizeof( unsigned int) * 8
unsigned int rol(), ror();

main()
{
        unsigned int num;
        printf("Introduceți un număr hexa: ");
```



```
        scanf("%x", &num);
        printf("rotire stînga 8 biți = %x\n", rol( num, 8));
        printf("rotire dreapta 4 biți = %x\n", ror( num, 4));
}

unsigned int rol( unsigned int word, int n)
{
        return word << n | ( word >> CUVDIM - n);
                        }

unsigned int ror( unsigned int word, int n)
{
        return word >> n | ( word << CUVDIM - n);
}
```

Execuția programului afișează:

```
>rot
Introduceói un numÑr hexa: abcd
rotire st¼nga 8 biói = cdab
rotire dreapta 4 biói = dabc
```

Prin utilizarea operatorului *sizeof* constanta simbolică *CUVDIM* este inițializată la numărul de biți ai tipului *unsigned int*.

Funcțiile de rotire spre dreapta și stînga sînt *ror* și *rol*. Nici una din funcții nu modifică valoarea originală a numărului.

Rotiri mai mari ca numărul de biți dat de *CUVDIM* produc rezultate incorecte. Se poate completa programul cu un test pentru evitarea acestei situații (a se vedea problema 1).

1.1.4. Operatori pe pointer

Operatorul *&* : reîntoarce adresa variabilei pe care o precede și poate fi gîndit ca "adresa lui".

Operatorul * : preia valoarea variabilei pe care o precede și utilizează această valoare ca adresă a informației în memorie. Poate fi gîndit ca "la adresa".

Pointeri de tip void sînt pointeri generici și pot fi utilizați pentru a pointa la un obiect de orice tip. În consecință, pointeri de orice tip pot fi atribuiți la pointeri de tip void si viceversa, prin folosirea unui cast de tip.

Declararea unui astfel de pointer este:

```
void *p;
```

şi se utilizează de obicei cînd o aceeaşi funcţie manipulează diferite tipuri de pointeri.

1.1.5. Operatori de atribuire

Operatorul de atribuire este =.
Expresii de forma:

```
x = y = z = 7;
```

sînt permise deoarece ordinea de evaluare (atribuire) este de la dreapta la stînga. În cazul în care se doreşte modificarea unei variabile cu valoarea unei expresii se permite ca în locul atribuirii:

```
var = var op expresie;
```

să se folosească o atribuire compusă:

```
var op= expresie;
```

unde *op* poate fi unul din operatorii: *, /, %, +, -, >>, <<, ^, !, &, |.
Operanzii unui operator de asignare compusă trebuie să fie de tip întreg sau flotant. Operatorii += şi -= pot avea operandul stîng de tip pointer, în care caz operandul drept trebuie să fie de tip întreg.

În atribuiri, tipul valorii din partea dreaptă este convertit la tipul valorii din partea stîngă.

1.1.6. Operatori de tip mixt

1.1.6.1. Operatorul condiţional (? :)

Operatorul condiţional are sintaxa:

```
expr1 ? expr2 : expr3
```


Dacă *expr1* (în general este o condiţie) se evaluează la o valoare diferită de zero (este adevarată), atunci rezultatul este dat de evaluarea expresiei *expr2*, în caz contrar rezultatul este dat de evaluarea expresiei *expr3*. Operatorii condiţionali pot fi imbricaţi.

Fie *c* o variabilă caracter iniţializată cu codul ASCII al unui caracter din intervalul *[0, 9]* sau *[a, f]*. Expresia:

```
c >= '0' && c <= '9' ? c - '0': c -('a' - 10)
```

va converti caracterul în număr zecimal. Acest tip de expresii este util în argumente de funcţii, instrucţiuni *return* şi macrouri.

1.1.6.2. Operatorul virgulă

Se mai numeşte operatorul de evaluare secvenţială şi se utilizează cel mai adesea în cadrul instrucţiunii *for*. Exemplu:

```
for (i=j=1; i+j<100; i+=2,j--);
```

A nu se confunda operatorul virgulă cu separatorul virgulă. Cînd virgula este utilizată în partea dreaptă a unei instrucţiuni de atribuire, valoarea întregii expresii este dată de valoarea ultimei expresii din lista de expresii separate prin virgulă. De exemplu, după execuţia instrucţiunilor:

```
y=10;
x=(y=y-5,30/y);
```

valoarea variabilei *y* este *5*, iar valoarea lui *x* este *6*.

1.1.6.3. Operatorul sizeof

Deşi poate fi considerat cuvînt cheie, operatorul este utilizat pentru a determina mărimea în octeţi a unei variabile sau unui tip de dată. Forma de utilizare este:

```
sizeof ( expresie)
```

unde *expresie* poate fi identificator sau orice tip de dată, mai puţin tipul *void*.

1.1.7. Precedența, asociativitatea și comutativitatea operatorilor

Precedența și asociativitatea operatorilor limbajului C este dată în tabelul de mai jos. Operatorii sînt listați în ordine descrescătoare a precedenței cu următoarele precizări:

- cînd mai mulți operatori apar în același rînd sau în paranteza dreaptă din stînga tabelului ei au aceeași precedență.
- toți operatorii unari au aceeași precedență.
- toți operatorii de atribuire simplă sau compusă au aceeași precedentă.

Simbol	Tip operatie	Asociere
() [] . ->	expresie	de la stg.la dr
- ~ ! * &	unara	de la dr.la stg
++ -- sizeof cast	unara	de la dr.la stg
* / %	multiplicativ	de la stg.la dr
+ -	aditiva	de la stg.la dr
<< >>	depalsare	de la stg.la dr
< > <= >=	relationala(ineg.)	de la stg.la dr
== !=	relationala(egalit.)	de la stg.la dr
&	SI pe biti	de la stg.la dr
^	SAU exclusiv pe biti	de la stg.la dr
|	SAU pe biti	de la stg.la dr
&&	SI logic	de la stg.la dr
||	SAU logic	de la stg.la dr
?:	conditionala	de la dr.la stg
= *= /= %= += -= <<= >>= &= |= ^=	atribuire simpl si compus	de la dr.la stg
,	evaluare secvential	de la stg.la dr

Tabelul 1.5.

În ceea ce privește expresiile ce conțin operatori comutativi (*, +, &, ^, |) acestea sînt evaluate de către compilator într-o ordine arbitrară, determinată de rațiunea optimizării codului generat.

1.2. Instrucțiuni de control a fluenței programului

Sfîrșitul fiecărei instrucțiuni se marchează obligatoriu prin caracterul ;. Limbajul C permite utilizarea instrucțiunii vide în orice loc, unde este permisă o instrucțiune. Acoladele definesc un *bloc* care poate conține la început declarații de variabile. Domeniul de existență al unei variabile este blocul care conține declarația.

1.2.1. Instrucțiunea if și if-else.

Instrucțiunea *if* poate avea trei formate de utilizare. Pentru fiecare din ele delimitatorii de bloc sînt opționali dacă există o singură instrucțiune. Primul ar fi:

```
if ( expresie) {
        instrucțiuni;
}
```

Dacă *expresie* este evaluată la o valoare diferită de zero, se execută *instrucțiuni*, iar în caz contrar, se execută instrucțiunea ce urmează blocului. Al doilea format este:

```
if ( expresie) {
        instrucțiuni1;
}
else {
        instrucțiuni2;
}
```

Dacă *expresie* se evaluează la o valoare diferită de zero, se execută *instrucțiuni1*, în caz contrar *instrucțiuni2*. Ultimul format:

```
if ( expresie1) {
        instrucțiuni1;
}
else if ( expresie2) {
        instrucțiuni2;
...
else ( expresie3) {
        instrucțiuni3;
```

permite utilizarea imbricată *if-else*. Ultimul *else* este opțional și se execută doar dacă expresiile anterioare sînt evaluate la valoarea zero. Întotdeauna un *else* se asociază cu ultimul *if* ce nu conține *else*.

Pentru exemplificare, fie *c* o variabilă de tipul caracter, inițializată cu o valoare hexazecimală, din intervalul *[0, 9]*, *[a, f]* sau *[A, F]* (cod *ASCII*). Următoarele instrucțiuni de control *if-else* convertesc caracterul *c* la un număr zecimal și afișează rezultatul.

```
if ( c >= '0' && c <= '9')
        printf("%d\n", c-'0');
else if ( c >= 'a' && c <= 'f')
        printf("%d\n", c-( 'a'- 10) );
else if ( c >='A' && c <='F')
        printf("%d\n", c-( 'A'- 10) );
else
        printf("Caracter hexazecimal invalid\n");
```

Dacă *c* este caracterul *'8'* programul afișează cifra *8*, iar dacă *c* este caracterul *'e'* sau, programul afișează cifra *14*. Dacă *c* nu conține codul *ASCII* al unui caracter din intervalul de mai sus se afișează un mesaj de eroare.

1.2.2. Instrucțiunea while

Formatul unei instrucțiuni *while* este:

```
while ( expresie) {
        instrucțiuni;
}
```

Se evaluează *expresie* și atîta timp cît valoarea ei este diferită de zero se execută *instrucțiuni*. Dacă *expresie* se evaluează la valoarea *0* execuția instrucțiunii *while* se încheie. Ca atare, este posibil ca *instructiuni* să nu se execute niciodată (dacă *expresie* este evaluată la zero). Blocul poate lipsi dacă întreaga acțiune se realizează la evaluarea expresiei. De exemplu, instrucțiunea:

```
while (( c=getchar()) != ' ' && c != '\t' &&
                c != '\n' && c != EOF)
                        ;
```


are ca efect ignorarea oricărui caracter mai puțin a caracterelor: blanc, tab, linie nouă și sfîrșit de fișier.

1.2.3. Instrucțiunea do while

Formatul unei instrucțiuni *do while* este:

```
do {
        instrucțiuni;
} while ( expresie);
```

Instrucțiuni se execută atîta timp cît *expresie* se evaluează la o valoare diferită de zero. Deosebirea față de instrucțiunea *while* este că mai întîi se execută *instructiuni* și apoi se evaluează *expresie*, deci bucla se execută cel puțin o dată.

Bucla *do while* este utilă, în procese repetabile cînd se așteaptă introducerea unei date de la tastatură (luarea unei decizii). Spre exemplificare programul:

Programul 1.2.

```
#include <stdio.h>
#include <conio.h>
main()
{
        char c;
        do {
        printf("\nIntroduceti o cifra:");
        c=getche();
        } while (c<'0' || c>'9');
        printf("\tCodul ASCII este: %x h = %d\n", c, c);
}
```

afișează codul ASCII al unei cifre în format hexazecimal și zecimal. Bucla *do while* nu permite afișarea și terminarea programului atîta timp cît în locul unei cifre s-a tastat un caracter.

1.2.4. Instrucțiunea for

Formatul unei instrucțiuni *for* este:

```
for ( expresie1; expresie2; expresie3) {
        instrucțiuni;
}
```

Efectul ei rezultă din instrucțiunile:

```
expresie1;
while ( expresie2) {
        instrucțiuni;
        expresie3;
}
```

Expresie1 se evaluează o singură dată și constituie inițializarea variabilei buclei. Dacă *expresie2* (testul buclei) se evaluează la o valoare diferită de zero se execută *instrucțiuni*, în caz contrar se iese din buclă. *Expresie3* (pasul buclei) se execută doar dacă testul este adevărat și are ca scop modificarea (incrementarea sau decrementarea) variabilei buclei. Toate aceste expresii sînt opționale putînd apărea instrucțiunea:

```
for (;;)
```

care reprezintă o buclă infinită.

Un exemplu de utilizare a instrucțiunii *for* îl constituie conversia unui întreg în hexazecimal.

Programul 1.3.

```
/* itoh1.c - conversie întreg la hexazecimal */
        /* utilizînd bucla for */
#include <stdio.h>
main()
{
        unsigned int num, d;
        int i=0;
        char buf[ 10];
        printf("Introduceți numărul: ");
        scanf("%u", &num);
        for( ; num; num /= 16) {
                d = num % 16;
                buf[i++]=( d <= 9) ? d+'0' : d+'a'-10;
        }
```



```
        for (;i;)
                putchar( buf[ --i]);
        putchar('\n');
}
```

Prima bucla *for* face împărțirea întregului fără semn *num* la *16*, după memorarea restului împărțirii la *16* în variabila *d*. Se părăsește bucla dacă valoarea variabilei *num* devine zero. A doua buclă *for* tipărește rezultatul prin afișarea conținutului șirului *buf* în ordine inversă.

1.2.5. Instrucțiunea switch

Formatul instrucțiunii switch este:

```
switch (expresie) {
        case exprct1:
                instrucțiuni;
                break;
        case exprct2:
                instrucțiuni;
                break;
        ...
        default:
                instrucțiuni;
                break;
}
```

Exprct1, exprct2, etc sînt expresii care se evaluează la constante întregi în momentul compilării. În caz contrar se semnalează eroare. Compilatorul evaluează *expresie* și compară valoarea acesteia cu expresiile constante (*exprct1*, *exprct2*, ...). Expresiile constante pot fi numite etichete. Dacă valoarea expresiei coincide cu una dintre etichete se execută *instrucțiuni*le atașate *case*-ului selectat. Omiterea instrucțiunii *break* dintr-un *case* are ca efect trecerea la execuția instrucțiunilor *case*-ului următor. Cuvîntul cheie *default* este opțional și instrucțiunile sale se execută, dacă nici un *case* nu a fost selectat. Ordinea pentru *case* și *default* nu este importantă în corpul instrucțiunii switch. Dacă nici un *case* nu a fost selectat și *default* lipsește instrucțiunea *switch* este inoperantă.
TC permite folosirea operatorilor în expresiile constante. Un exemplu este:

Programul 1.4.

```
#define ROSU        1
#define GALBEN      2
#define ALBASTRU    4

#include <stdio.h>

main()
{
        int i;
        for ( i=1; i<8; i++)
        switch( i) {
                case ROSU:
                        printf("rosu\n");
                        break;
                case GALBEN:
                        printf("galben\n");
                        break;
                case ALBASTRU:
                        printf("albastru\n");
                        break;
                case ROSU+GALBEN:
                        printf("portocaliu\n");
                        break;
                case ROSU+ALBASTRU:
                        printf("violet\n");
                        break;
                case ALBASTRU+GALBEN:
                        printf("verde\n");
                        break;
                default:
                printf("Culoare necunoscută\n");
        }
}
```

Operatorul +, din *case* formează culorile secundare (portocaliu, verde și violet), din culorile principale (roșu, galben și albastru). Utilizarea operatorului este permisă, deoarece expresiile se evaluează la constante întregi.

Implementarea instrucțiunii *switch* este dependentă de compilator. Codul generat în asamblare este în funcție de numărul și valoarea etichetelor

case. Dacă numărul de *case*-uri este mic compilatorul tratează instrucțiunea *switch* ca o instrucțiune *if* imbricată. Dacă numărul de *case*-uri este mare și etichetele au valori succesive (de exemplu *1,2,...,15*) compilatorul construiește o tabelă de salt. În acest caz, instrucțiunii *switch* i se atribuie o adresă de bază, iar etichetele reprezintă deplasamente. Dacă însă valoarea etichetelor este împrăștiată (de exemplu *20,40,60* etc) se generează un set de perechi valoare-adresă pentru saltul la etichetele selectate.

Acest lucru explică motivul pentru care C-ul permite doar expresii întregi în expresiile constante. Varianta cu tabelă de salt are avantajul că timpul de execuție pentru saltul la orice *case* este același, pe cînd în varianta *if-else* el depinde de numărul de expresii care se evaluează.

1.2.6. Instrucțiunile break și continue

Instrucțiunile *break* și *continue* apar în interiorul buclelor sau în instrucțiuni *switch*. Instrucțiunea *break* încheie prematur o secvența de instrucțiuni, iar instrucțiunea *continue* forțează reluarea imediată buclei (o nouă iterație). Ambele instrucțiuni afectează doar bucla ce le conține.
Instrucțiunea *break* se poate utiliza pentru a ieși forțat dintr-o buclă *for* infinită. Astfel, dacă *c* este o variabilă de tip întreg, atunci codul:

```
for(;;) {
        printf("Selecție: ");
        if( ( c = getche()) == 'q')
                break;
        meniu(c);
}
```

permite părăsirea buclei dacă se tastează caracterul *'q'*. În caz contrar se apelează funcția *meniu()*.
Următoarul segment de cod, contorizează în variabila *ncar* numărul de caractere ce se introduc de la tastatură, mai puțin caracterul *ESC*.

```
for ( ncar = 0;;){
        printf("Introduceți un caracter: ");
        switch (getchar()) {
                case '\033':        /* ESC */
                        continue;
                case '\n':
                        break;
```

```
        ...
    }
    ncar++;
    ...
}
```

Pentru claritate programele pot evita utilizarea instrucțiunii *break* și *continue* printr-o buclă *do while*.

1.2.7. Instrucțiunea goto

Formatul unei instrucțiuni *goto* este:

```
goto etichetă;
```

cu specificarea că *etichetă* trebuie să apară în aceeași funcție în formatul:

```
etichetă: instrucțiuni;
```

Efectul instrucțiunii *goto* este saltul (intern blocului funcției) la instrucțiunea cu eticheta specificată. De regulă, această instrucțiune se evită în programarea structurată, prin diverse artificii de pogramare. Se recomandă utilizarea ei în programe în care timpul este esențial.
Saltul între funcții este posibil prin utilizarea funcțiilor *setjmp()* și *longjmp()*, care vor fi prezentate în capitolul 2.

1.3. Șiruri și pointeri

Șirul sau tabloul este o mulțime ordonată de date de același tip. Dimensiunea tabloului este dată de numărul inidicilor necesari pentru referirea unui element al tabloului. Ca atare, vectorii sînt tablouri unidimensionale, iar matricile sînt tablouri bidimensionale. Formatul declarației unui șir unidimensional este:

```
tip numetablou[ dim] = { listă de inițializare};
```
unde
- *tip* este un tip de dată,
- *dim* este numărul de elemente ale șirului,
- *lista de inițializare* este o listă de valori.

Lista de inițializare este opțională, numărul valorilor ei fiind mai mic sau egal cu valoarea *dim*. Precizarea valorii *dim* este redundantă dacă lista de inițializare actualizează toate elementele șirului.

Referirea la un element al tabloului se face folosind numele tabloului și un număr de indici. Pentru un tablou unidimensional formatul este:

```
numetablou[ expresie]
```

unde: *expresie* trebuie să fie o expresie întreagă. Indicele primului element din șir este zero, iar al ultimului este cu unu mai mic decît dimensiunea șirului.

Declarații corecte de șiruri sînt:

```
char buf[ 80];
char tab[]={ 'c','o','r','e','c','t','\0'};
double șir[3]={ 6.23, -4.7};
```

Buf este un tablou neinițializat de *80* de caractere, *tab* un șir de șapte caractere, iar *șir* un tablou de trei elemente de tip double (primele două elemente avînd valorile specificate).

Inițializarea unui șir de caractere se poate face mai simplu prin declarația:

```
char tab[]="corect";
```

unde compilatorul include automat octetul terminator pentru șirurile de caractere (*NULL*).

Tabloul (șirul) multidimensional este un tablou unidimensional ale cărui elemente sînt tablouri. Asupra tablourilor multidimensionale vom reveni în capitolul 3.

1.3.1. Conceptul de pointer

Prin acestă noțiune limbajul C este unul dintre cele mai puternice limbaje de programare. Un pointer este o variabilă care reprezintă o adresă de memorie. Prin această adresă se pot accesa sau referi obiecte din memorie, fiind ca atare un mecanism de acces indirect. Declarația unui pointer utilizează simbolul *, pentru a marca faptul că respectiva valoare este o adresă. Dacă operatorul de indirectare sau dereferențiere (*) se aplică unei variabile ce conține o adresă, se obține obiectul pe care-l pointează sau îl referă pointerul. Cel mai simplu exemplu este atribuirea adresei unei variabile.

```
char c;
p = &c;
```

Operatorul unar *&* (ampersand) aplicat unui obiect întoarce adresa lui. Pointerul este întotdeauna asociat cu un tip de dată, sau altfel spus orice pointer referă un tip de dată. Ca atare, obiectul este o dată ce are asociat un tip. Pentru exemplul de mai sus pointerul *p* referă o variabilă de tipul caracter.

Prima regulă de care trebuie ţinut cont la lucrul cu pointeri este declararea pointerilor. Formatul unei declaraţii este:

```
tip * pobiect;
```

pentru un pointer la tipul *tip*, sau

```
tip * pşir[ dim1][ dim2][...]
```

pentru un pointer la un şir de tipul *tip*. Revenind la exemplul anterior, se impune ca necesară declaraţia:

```
char *p;
```

Tipul pointerului permite compilatorului să cunoască dimensiunea datelor referite de acel pointer. Aritmetica cu pointeri utilizează această observaţie ca regulă de bază.

Un pointer de tipul *void* (fără tip) serveşte la memorarea adresei oricărui tip de variabilă. Acest tip de pointer nu se poate dereferenţia.

A doua regulă în lucrul cu pointeri, se referă la iniţializarea oricărui pointer înainte de utilizare. În caz contrar, ei referă ceva arbitrar din memorie.

Un pointer poate fi iniţializat în două moduri.

Primul mod ar fi prin atribuirea adresei unui obiect existent deja în memorie. În aceste cazuri se foloseşte operatorul *&*, pentru găsirea adresei obiectului. Acest operator nu este necesar la aflarea adresei unui şir sau unei funcţii, deoarece compilatorul converteşte implicit numele şirului şi funcţiei la adresa primului element din şir respectiv la adresa funcţiei.

Al doilea mod de a iniţializa un pointer este printr-un proces de alocare dinamică. În momentul execuţiei, programul rezervă o zonă de memorie şi returnează prin funcţiile de alocare adresa de început a acestei zone, care se atribuie pointerului. Pentru a schimba tipul pointerului returnat de funcţiile de alocare se foloseste castul.

Exemple de declaraţii şi iniţializări valide pentru lucrul cu pointeri sînt:


```
struct complex {
        double real, imag;
};
char şir[10], *ptr_şir;
char c='a', *ptr_char;
int i=1, *ptr_int = &i;
struct complex z = { 1.41, 1.72}, *ptr_cplx;
ptr_char = &c;
ptr_cplx = &z;
*ptr_int = 0; /* schimbă conţinutul lui i la 0 */
ptr_sir = şir;
```

O declaraţie de tipul:

```
tip * ptr_tip;
```

se poate interpreta ca:

ptr_tip este un pointer la tipul *tip* sau
**ptr_tip* este un caracter.

Ca reprezentare internă pointerul este reprezentat pe doi octeţi, excepţie făcînd pointerii de tip *far*, care sînt reprezentaţi pe patru octeţi.

Se recomandă folosirea pointerilor peste tot unde natura problemei permite acest lucru (în locul indexării - capitolul 3). Un cod ce foloseşte pointeri este lizibil, compact şi foarte eficient.

Conceptul de pointer se poate extrapola la cel de pointer la pointer ş.a.m.d.

1.3.2. Pointerul ca argument în funcţii

Particularitatea transmiterii argumentelor la funcţii prin valoare (call by value) implică imposibilitatea modificării directe a parametrilor actuali ai funcţiei. În situaţiile cînd se doreşte modificarea conţinutului unor variabile trebuie transmisă funcţiei adresa variabilei şi nu variabila propriu-zisă. În acest caz se spune că apelul este prin referinţă (call by adress).

Exemplul tipic pentru un astfel de caz este interschimbarea conţinutului a două variabile printr-o funcţie *swap()*. Rescriem funcţia dată de *K & R* în maniera:

Programul 1.5.

```
/* swap.c - interschimb conținutului variabilelor a și b */

#include <stdio.h>

main()
{
        int a = 77, b = 11;
        swap( &a, &b);
        printf("a= %d b= %d\n", a, b);
}

swap( int *pa, int *pb)
{
        *pa ^ = *pb;
        *pb ^ = *pa;
        *pa ^ = *pb;
}
```

Rezultatul executiei este: a= 11 b= 77.

Dacă funcției *swap()* nu i s-ar transmite adresele variabilelor *a* și *b*, valoriile lor ar rămîne nemodificate la revenire în funcția *main()*. Aceasta din cauza că funcția *swap()* nu ar avea acces la variabilele din apel ci numai la valorile lor. În cazul de față, funcției *swap()* i se transmit adresele variabilelor *a* și *b* a căror conținut este interschimbat. Acest aspect este observabil în multe din programele ce urmează.

1.4. Argumentele liniei de comandă

Linia de comandă este zona de introducere si editare a comenzii. Formatul general al unei linii de comandă este:

```
c:\> nume-program argument1 argument2 ...
```

Numărul argumentelor liniei de comandă poate varia funcție de necesitățiile programului.

Limbajul C permite scrierea de programe, care utilizează argumentele liniei de comandă. În acest caz, funcția principală are sintaxa:

```
main( int argc, char *argv[])
```


unde,

argc - reprezină numărul de argumente din linia de comandă, inclusiv numele programului,

argv - este un șir de pointeri de tip caracter la argumentele liniei de comandă.

Tabelul de mai jos ilustrează argumentele liniei de comandă referite de fiecare element a lui *argv*:

i	Continutul lui argv[i]
0	numele programului
1	argumentul 1
2	argumentul 2
...	
argc-1	argumentul argc-1

Tabelul 1.6

În TC, pentru programele ce utilizeaza linia de comandă, argumentele se pot preciza direct din mediul TC, prin selectarea opțiunii "Options", din meniul principal, iar apoi prin editare în fereastra obținută după selectarea opțiunii "Arguments".

De exemplu, fie programul **cmd.c** care permite două opțiuni în linia de comandă. Fie acestea */p* și */w*. Pentru linia de comandă:

```
>cmd /p
```

variabila *argc* are valoarea *2*, iar pointerii *argv[0]* și *argv[1]* referă șirurile "*com*" și "*/p*".

Segmentul de cod ce urmează afișează opțiunile valide ale programului **cmd.c**.

Programul 1.6.

```
/* cmd.c - argumentele liniei de comandă */
#include <stdio.h>
```

```
main( int argc, char *argv[])
{
    if (argc != 2) {
        fprintf( stderr,"Utilizare: %s /[pw]\n", argv[0]);
        exit(1);
    }
    switch ( argv[1][0]) {
        case '/':
            switch ( argv[1][1]) {
                case 'p':
                    printf("opțiunea /p");
                    break;
                case 'w':
                    printf("opțiunea /w");
                    break;
                default:
                    fprintf(stderr,"opțiune ilegală pentru %s\n", argv[0]);
                    exit(1);
            }
            break;
        default:
            fprintf( stderr,"opțiune nespecificată pentru %s\n", argv[0]);
            exit(1);
    }
}
```

Dacă *argc* este diferit de *2* programul se termină cu afișarea unui mesaj de informare asupra modului de utilizare. În caz contrar, instrucțiunea *switch* analizează opțiunea din linia de comandă. Dacă primul caracter al opțiunii este *'/'* se verifică dacă următorul caracter este *'p'* sau *'w'* și se afișează opțiunea identificată. În cazul în care al doilea caracter nu este nici *'p'* nici *'w'*, sau dacă opțiunea nu începe cu caracterul *'/'* se afișează cîte un mesaj de eroare.

Execuția programului pentru diferite linii de comandă afișează:

```
>com
Utilizare: cmd.exe /[pw]
>com /o
opțiune ilegală pentru cmd.exe
>com /p
opțiunea /p
```


1.5. Funcții

Un program C constă dintr-o listă de declarații și definiri de funcții. Orice program C conține cel puțin funcția *main()*. Limbajul C nu admite definiția funcțiilor imbricate.

Funcția este o colecție independentă de declarații și instrucțiuni, realizată în scopul rezolvării unei sarcini propuse. Definiția unei funcții specifică numele, tipul și clasa de memorie a funcției, numărul și tipul parametrilor formali, declarațiile și instrucțiunile ce realizează acțiunile funcției (corpul funcției), precum și tipul rezultatului returnat de funcție. Definiția unei funcții trebuie să conțină:

1) declarația și
2) definirea sau corpul funcției.

1.5.1. Declarația funcțiilor

O declarație de funcție face cunoscută programului o funcție. Această funcție poate fi utilizată de celelalte funcții din program.

Declarația unei funcții are același format ca și definirea funcției cu excepția faptului că îi lipsește corpul. Sfîrșitul unei declarații este marcat prin caracterul *';'*. Declarația funcției precede întotdeauna definirea ei. Declarația stabilește caracteristicile unei funcții astfel încît apelurile ulterioare (ce se pot găsi în alt fișier decît cel ce conține declarația) pot fi verificate din punctul de vedere al compatibilității tipului argumentelor și rezultatului returnat.

Declarația unei funcții specifică, pe de o parte:

- numele funcției,
- clasa de memorie a funcției și tipul valorii returnate;

iar pe de altă parte:

- declarația parametrilor formali.

Tipul valorii returnate este implicit întreg. Dacă o funcție nu returnează nimic util pentru program, se obișnuiește specificarea tipului *void*. Clasa de memorie pentru funcții este implicit *extern*. Declarația parametrilor include tipul și numărul parametrilor formali. Opțional, se poate include și numele parametrilor formali.

Există două stiluri de declarare a funcțiilor: stilul clasic și stilul modern. Ele sînt ilustrate mai jos:

```
a)      double func(int i,double *p);      /* stil modern */
        double func(int,double *);                 /* condensat */
b)      double func();                     /* stil clasic */
```

În cazul a) se specifică numele funcției, tipul și opțional numele fiecărui parametru formal. Acest stil de declarație permite compilatorului să verifice numărul și tipul parametrilor actuali din apelul funcției. Dacă este posibil compilatorul face conversiile de tipuri.
În cazul b) se specifică doar numele funcției și tipul valorii returnate fără informații despre parametrii formali. În acest caz compilatorul nu face nici un fel de verificări.

Declarațiile de funcții permit compilatorului diagnoza și tratarea corectă a neconcordanței tipurilor și numărului de argumente, care altfel se observă, eventual, doar la execuție.

La lucrul cu funcții trebuie ținut cont de două considerente:

a) compilatorul nu convertește implicit parametrii funcției de la tipul ***float*** la tipul ***int*** și invers.
b) dacă o funcție returnează un tip de dată, altul decît ***int***, compilatorul trebuie informat asupra tipului valorii returnate înainte de apelul funcției.

Ilustrăm aceste două considerente pe următoarele exemple:

Programul 1.7.

```
/* sqrt1.c : transmiterea incorectă a parametrilor */
main()
{
        printf("%.8g\n", sqrt( 25));
}
```

Execuția programului afișează:

1.6173145e-232

Programul 1.8.

```
/* sqrt2.c - transmiterea incorectă a parametrilor */
main()
{
        printf("%.8g\n", sqrt( 25.));
}
```


Execuția programului afișează:

1.6173145e-232

Programul 1.9.

```
/* sqrt3.c - transmiterea corectă a parametrilor unei funcții */

#include <math.h>

main()
{
        printf("%.8g\n", sqrt( 25));
}
```

Execuția programului afișează:
5

Funcția ***sqrt()*** returnează o valoare de tip ***double*** și necesită tipul ***double*** la argument. Fiecare din cele trei programe calculează radicalul din 25, dar numai programul **sqrt3.c** afișează rezultatul corect.

Programul **sqrt1.c** transmite ca parametru o dată de tip ***int***. Compilatorul, în concordanță cu considerația *a)*, generează cod în limbaj de asamblare, care interpretează întregul ca un număr de tipul ***double***. Acest lucru generează rezultatul incorect.

Programul **sqrt2.c** transmite funcției *sqrt()* un parametru valid (numărul real 25). Valoarea returnată de funcția *sqrt()* este interpretată incorect, deoarece compilatorul în concordanță cu regula *b)*, generează cod în limbaj de asamblare pentru tipul *int* în loc de tipul ***double***. Programul trebuie să specifice compilatorului tipul ***double***, ca tip returnat de funcția *sqrt()*. Astfel, tipul implicit fiind întreg, funcția *printf()* afișează un întreg cu descriptorul de format "*%.8g*" (utilizat pentru *float*).

Programul **sqrt3.c** include fișierul **math.h**, ce are ca efect utilizarea corectă a funcției *sqrt()*. Acest fișier include declarația:

```
double sqrt( double);
```

care specifică tipul așteptat al rezultatului. În declarație este suficientă specificarea tipului argumentului, în conformitate cu cele discutate anterior.

Compilatorul utilizează declarația funcției pentru a face corespondența tipului parametrilor formali cu tipul parametrilor actuali din apelul funcției. Corespondența trebuie să fie biunivocă, în caz contrar se semnalează la compilare eroare de neconcordanță de tipuri

1.5.2. Definirea funcțiilor

Definirea unei funcții include numele, parametrii formali, tipul valorii returnate și corpul funcției. Corpul funcției conține (între acolade) declarații de variabile locale, referiri externe și instrucțiuni.

Formatul unei definiri de funcție este:

```
tipret func( tip1 arg1, tip2 arg2, ...)
{
        /* declarații locale */
        ...
        /* instrucțiuni */
}
```

Tipul *tipret* al valorii returnate este opțional, implicit este tipul ***int***. Astfel acest tip poate fi orice tip de dată acceptat de limbajul C sau cuvîntul cheie ***void*** (utilizat cînd funcția nu returnează ceva util programului). Utilizarea tipului ***void*** la definirea unei funcții nu permite ca valoarea returnată de funcție să apară într-o expresie (în caz contrar se semnalează eroare la compilare). Argumentele *arg1*, *arg2*, ... sînt opționale (funcție fără argumente). Pentru o funcție ce nu necesită argumente se poate preciza explicit, între paranteze, tipul ***void***.

Dacă o funcție are argumente, tipul lor trebuie precizat. Omiterea tipului cînd acesta este diferit de *int* (tipul implicit) cauzează execuții eronate. De exemplu, dacă o funcție are declarația:

```
long func( număr)
{
        ...
}
```

un apel de genul:

```
long i=70000;
...
printf("%ld\n", func( i));
...
```


nu afișează rezultatul scontat, deoarece apelul se face cu parametrul actual *i*, care este de tipul *long*, iar parametrul formal *număr* este de tip *int* (implicit).

Pentru exemplele prezentate avem definirile:

```
double func1( int j, double *p)
{
        return( j+ *p);
}
```

sau

```
double func2( j, p)
int j; double *p;
{
        return ( j+ *p);
}
```

Nu există nici o legătură între numele parametrilor formali (din declarație) și cei actuali (din definire). De exemplu, *i* cu *j*.

1.5.3. Apelul funcțiilor

Apelul funcției transferă controlul de la funcția apelantă la funcția apelată. Apelul funcției se face prin numele funcției, urmat de lista reală a parametrilor, închisă între paranteze. Apelul, fiind instrucțiune, se termină cu caracterul *;*.

Dacă într-o expresie numele funcției nu este urmat de paranteza stîngă, se generează un pointer la acea funcție.

Argumentele actuale, dacă ele există, sînt transmise funcției apelate prin valoare. Asupra acestor aspecte vom reveni în capitolul 4.

Se permit și apeluri de funcții cu număr variabil de argumente. Această tehnică este prezentată în capitolul 4.

1.5.4. Revenirea din funcții

Instrucțiunea de revenire din funcții este opțională și poate avea sintaxa:

```
return;
return expr;
return (expr);
```

Tipul valorii returnate este specificat în fața numelui funcției. Dacă el este omis tipul implicit este întreg.

1.5.5. Recursivitate. Funcții recursive

Un obiect se consideră a fi recursiv dacă se poate defini în funcție de el însuși. Puterea recursivității, ca instrument matematic, constă în posibilitatea de-a defini un set infinit de obiecte printr-un număr finit de reguli.

În felul acesta, se poate descrie un număr infinit de calcule printr-un program recursiv finit, chiar dacă acest program nu conține repetiții explicite.

C-ul permite apeluri recursive de funcții. La scrierea unei funcții recursive sînt esențiale două aspecte:

- condiția de terminare și
- numărul de apeluri recursive.

Se analizează aceste două aspecte pe cîteva exemple. Calculul factorialului constituie un exemplu simplu de recursivitate. Relația:

$$n ! = n (n-1)!$$

permite o rezolvare recursivă, deoarece se poate calcula factorialul unui număr *n*, dacă se cunoaște factorialul numărului *n-1*. Condiția de terminare este *0!=1* sau *1!=1*.

Numărul de apeluri recursive este limitat doar de dimensiunea memoriei disponibile, deoarece mecanismul recursivitații salvează mereu în stivă rezultatele parțiale. Aceste considerente conduc la programul:

Programul 1.10.

```
/* rec1.c - Calculul recursiv al factorialului */
#include <stdlib.h>
int factorial( int n)
{
        if ( n < 0) {
                printf("Nu se poate calcula factorialul\n");
                exit(0);
        }
        if ( n==1)
                return 1;
        return n*factorial( n-1);
}
```



```
main( int argc, char *argv[])
{
        int fact;
        if ( argc < 2) {
                printf("Utilizare: %s număr\n", argv[0]);
                exit(0);
        }
        if ( (fact= atoi(argv[1])) > 15) {
                printf("Depașirea capacitații de calcul !\n");
                exit(0);
        }
        fact=factorial( fact);
        printf( "Rezultat: %d", fact);
}
```

Exemplul fiind simplu, avantajul utilizării recursivității în acest caz este practic inexistent. Se constată totuși o simplificare a modului de gîndire a problemei.

Un alt exemplu îl constituie inversarea unui șir de caractere. Observînd că șirul de lungime unu nu trebuie inversat se găsește condiția de terminare. Șirul ce conține n caractere se poate inversa prin mutarea primului caracter la coadă șirului inversat de n-1 caractere.

Codul programului este:

Programul 1.11.

```
/* rec2.c - inversarea unui șir de caractere */
/*       utilizînd apelul recursiv de funcție */

#include <string.h>

char *top (char *p)
{
        return p;
}

char *rest( char *p)
{
        return ( p+1);
}
```

```
put_in( char *d, char *s)
{
        while ( *d++)
        ;
        *( d-1) = *s;
        *d = '\0';
}

rev( char *p, char *q)
{
        if ( strlen( p) == 1) {     /* condiția de terminare */
                strcpy( q, p);

/* o altă soluție pentru condiția de terminare

                strcpy( q, rest(p));
                put_in( q, top( p));
*/

        }
        else {
                rev( rest( p), q);
                put_in( q, top( p));
        }
}

main( int argc, char *argv[])
{
        char buf[25];
        if ( argc < 2) {
                printf("Utilizare: %s șirul_de _inversat\n", argv[0]);
                exit(0);
        }
        rev( argv[1], buf);
        printf("Rezultat: %s\n", buf);
}
```

Șirul de inversat este preluat din linia de comandă și este transmis ca argument funcției recursive *rev()*. Ea are doi parametrii formali: pointerul *p*, ce referă șirul de inversat și pointerul *q*, ce referă șirul inversat. Apelul recursiv al funcției *rev()* elimină, prin deplasarea pointerul *p* în șir, cîte un caracter din șir

pînă cînd șirul ajunge de un caracter. Se copiază acest caracter în șirul referit de pointerul *q* și se revine la inversarea șirului referit de *p*, care conținea două caractere. Se copiază primul caracter din șirul referit de *p* la coada șirului referit de *q*. În acest moment, șirul referit de *q* conține două caractere. Pașii se repetă pînă cînd se termină fiecare apel al funcției *rev()*.

Funcția *top()* returnează adresa primului element din șir, iar funcția *rest()* adresa subșirului rezultat prin omiterea primului element. Funcția *put_in()* copiază primul caracter din șirul de inversat pe prima poziție liberă din șirul inversat.

Recursivitatea fiind greu de urmărit se sugerează modificarea funcției *rev()* din programul anterior în maniera:

```
rev( char *p, char *q, int h)
{
        int i;
        for (i=0; i<h; i++)
                printf(" ");
        printf("Apel recursiv %d cu p: %s\n" ,h++,p);
        if ( strlen( p) == 1)
                strcpy( q, p);
        else {
                rev( rest( p), q, h);
                put_in( q, top( p));
                for (i=0; i<h; i++)
                        printf(" ");
                printf("Sfîrșit apel recursiv %d cu q: %s\n", h,q);
        }
}
```

Execuția programului afișează:

```
>rec2 "adrian"
```

afișează:

```
Apel recursiv 1 cu p: adrian
        Apel recursiv 2 cu p: drian
                Apel recursiv 3 cu p: rian
                        Apel recursiv 4 cu p: ian
                                Apel recursiv 5 cu p: an
                                        Apel recursiv 6 cu p: n
```

```
                                        Sfirsit apel recursiv 6 cu q: na
                                    Sfirsit apel recursiv 5 cu q: nai
                                Sfirsit apel recursiv 4 cu q: nair
                            Sfirsit apel recursiv 3 cu q: naird
                        Sfirsit apel recursiv 2 cu q: nairda
Rezultat: nairda
```

Se ilustrează nivelul de imbricare și conținutul celor două șiruri.

O altă problemă ce permite o abordare recursivă este sortarea unui șir. Se folosește algoritmul de sortare descoperit de C.A.R.Hoare în 1962. Fiind dat un șir, se alege un element arbitrar din el, cu care se compară apoi restul elementelor. Se obțin asfel două subșiruri, unul ce conține elementele mai mici și altul ce conține elementele mai mari comparativ cu elementul ales. Această tehnică se aplică recursiv subșirurilor. Condiția de terminare se obține în cazul în care subșirul conține două elemente. Versiunea implementată mai jos, utilizează apelul recursiv în funcția *qsort()*. Iată codul ei:

Programul 1.12.

```
/* rec3.c - sortarea unui șir utilizînd o funcție recursivă */
#include <string.h>

void qsort( int *, int *, int *);
void swap( int *, int *);

prt( int *p, int *d)
{
        int i, *pi;
        for ( pi=p; pi<=d; pi++)
                printf("%4d", *pi);
        printf("\n");
}

void qsort( int *p, int *s, int *d)
{
        int *sant;
        int *pi;
        pi = p;
        if ( s >= d)
                return;
        sant = s;
        for ( pi=s+1; pi <= d; pi++)
                if ( *pi < *s)
                        swap( ++sant, pi);
        prt( p, d);
        swap( s, sant);
        prt( s, sant-1);
        qsort( p, s, sant-1);
        prt( sant+1, d);
        qsort( p, sant+1, d);
}

main()
{
        int i;
        ...
/*      a       - șirul de întregi
        s       - pointerul la primul element al șirului
        d       - pointer la ultimul element al șirului
*/
        qsort(a, s, d);
        for ( i=0; i <= l; i++)
                printf(" %4d", a[i]);
        printf("\n");
}
```

Am utilizat pointerul *sant* (santinela), pentru a memora elementul cu care se face compararea. În cazul în care sortarea implică o interschimbare de elemente se apelează funcția *swap()*. Afișarea subșirurilor parțial ordonate și a șirului ordonat se realizează prin apelul funcției *prt()*.

Apelul funcției *qsort()* se face cu parametrii: adresa șirul ce se ordonează și adresele de început și sfîrșit ale șirului.

Recursivitatea nu e mai rapidă, dar codul rezultat este mai compact și deseori mai ușor de conceput și înțeles decît codul echivalent nerecursiv.

Recursivitatea este foarte utilizată la obiecte ce se pretează la o definire recursivă, cum ar fi listele și arborii.

Rezolvarea recursivă a celebrei probleme a turnurilor din Hanoi este prezentată în cele ce urmează:

Programul 1.13.

```
/* hanoi.c - problema turnurilor din Hanoi */
/*        rezolvare recursivă */


#include <stdio.h>

void hanoi( char, char, char, int);

main ( void )
{
        hanoi('A','B','C',3);
}

void hanoi( char a, char b, char c, int  n )
{
        if( n==1 )
                printf("se mută de pe discul %c pe discul %c\n", a ,b );
        else {
                hanoi( a, c, b, n-1);
                printf("se mută de pe discul %c pe discul %c\n", a, b);
                hanoi( c, b, a, n-1);
        }
}
```

Pentru ilustrarea conținutul turnurilor după fiecare mutare problema 16 oferă o posibilă rezolvare.

1.5.6. Nedeterminism și revenire (backtracking)

În practică există anumite tipuri de probleme a căror soluție nu este determinabilă pe bază unor calcule cunoscute, ci pe baza unor metode riguroase de încercări. Este ca și cum v-ați afla într-un labirint și ați încerca să ieșiți din el. În acest caz sînteți nevoiți să găsiți calea spre ieșire prin încercări. Cînd ajungeți la o ramificare de căi luați o decizie, despre care, în momentul respectiv nu veți putea afirma că este corectă sau incorectă. Dacă ea vă conduce spre ieșire înseamnă că a fost corectă, altfel nu. Este însă clar că dacă vă puteți da seama că ați luat o decizie incorectă și reușiți să vă întoarceți la locurile de ramificare, pentru a lua o altă decizie, mai devreme sau mai tîrziu veți ieși din labirint (dacă

el are soluție!). Dar tocmai aceasta e dilema, dificultatea întoarcerii la locurile de ramificare.

Revenirea prin nedeterminare, identică cu întoarcerea la locurile de ramificare din labirint, constă în reluarea algoritmului din locurile din care mai există căi de găsire a soluției. Dacă problema admite mai multe soluții și dorim să le găsim pe toate folosim, deasemenea, revenirea prin epuizarea tuturor căilor de căutare. Conștientizarea cît mai rapidă a momentului blocării algoritmului, adică oportunitatea revenirii rămîne apanajul programatorului rafinat.

Maniera de lucru este riguroasă, deoarece ea parcurge toate posibilitățile, selectînd în final soluția sau soluțiile, dacă ele există.

În astfel de probleme, recursivitatea este metoda cea mai convenabilă. Soluția unei astfel de probleme se obține din soluții parțiale, rezultate din apeluri recursive, care se modifică dinamic și se întregesc.

Avantajul folosirii recursivității rezidă în posibilitatea revenirii la locurile de luare a deciziei prin însuși mecanismul recursivității. Construirea unei funcții recursive adecvate, reduce considerabil activitatea de căutare și scriere a codului.

Pentru exemplificare se ilustrează problema calului. Ea constă din acoperirea completă a unei table pătratice de dimensiune dată prin mutări ale calului. Varianta ce găsește o primă soluție este ilustrată mai jos:

Programul 1.14.

```
/* caluț.c - soluționarea nedeterminismului */

# include <stdio.h>

int incerc(int,int,int);
void print(void);

# define N              5
# define NP             25
# define LIBER          0
# define SUCCES         1
# define ESEC  0

static int a[8] = {2, 1, -1, -2, -2, -1, 1, 2};
static int b[8] = {1, 2, 2, 1, -1, -2, -2, -1};
int h[N][N];
```

```
void main(void)
{
        int i,j;
        for (i=0; i<N; i++)
                for (j=0; j<N; j++);
                        h[i][j] = LIBER;
        h[0][0] = 1;
        if (incerc(2,0,0) == SUCCES)
                print();
        else
                printf("Problema nu are soluție\n");
}

int incerc(i,x,y)
{
        int k,xu,yu,q;
        for(k=0,q=ESEC; q==ESEC && k<8; k++) {
        xu=x+a[k];
        yu=y+b[k];
        if ( xu>=0 && xu<N && yu>=0 && yu<N)
                if ( h[xu][yu] == LIBER) {
                        h[xu][yu]=i;
                        if (i < NP) {
                                if ((q=incerc(i+1,xu,yu)) == ESEC)
                                        h[xu][yu] = LIBER;
                        } else
                                q = SUCCES;
                }
        }
        return(q);
}

void print()
{
        int i,j, k;
        for (k=0; k<N; k++)
                printf("-----");
        putchar('\n');
        for (i=0; i<N; i++) {
                for (j=0; j<N; j++)
                        printf("| %2d ",h[i][j]);
```



```
                printf("|\n");
                for (k=0; k<N; k++)
                        printf("-----");
                putchar('\n');
        }
}
```

Tabla este de dimensiune 5x5 și programul afișează prima soluție găsită sub forma unui tablou ce conține succesiunea mutărilor calului.

Evitați utilizarea recursivității dacă varianta echivalentă nerecursivă este evidentă și doriți să cîștigați timp.

Un exemplu concret de algoritm cu revenire îl constituie problema *selecției optimale*. O specificare particulară a problemei ar fi: se consideră obiecte cu două caracteristici: valoare și greutate. Să se caute acele obiecte a căror greutate totală nu depășește o valoare dată și a căror valoare totală este maximă. Algoritmul sugerat de N. Wirth respectă pseudocodul:

```
procedure punere( i: integer)
begin
        if (posibilă_introducerea) then
                begin
                        marcare_obiect;
                        if ( i<N) then
                                punere(i+1);
                        else
                                test_de_optim;
                        demarcare_obiect;
                end
        if (posibilă_eliminarea) then
                if (i<N)
                        punere(i+1);
                else
                        test_de_optim;
end
```

cu variabila *i* indice în șirul de obiecte.
Implementarea în C a acestei probleme este:

Programul 1.15.

```
#define N 10

typedef struct {
        int g;
        int v;
} OBIECT;

OBIECT a[N]={       10, 18,
                    11, 20,
                    12, 17,
                    13, 19,
                     4, 25,
                    15, 21,
                    16, 27,
                     7, 23,
                     8, 25,
                    19, 24
};

/*      glim    - greutatea limită a unui set
        maxv    - valoarea maximă a obiectelor din set
        totv    - valoarea totală a obiectelor
        s[N]    - şirul obiectelor selectate
        opts[N] - şirul optim de obiecte selectate */

int glim;
int maxv;
char s[ N], opts[ N];

incerc( int i, int gt, int av)

/*      i       - index în lista obiectelor
        gt      - greutatea curentă a obiectelor din set
        av      - valoarea totală a obiectelor neincluse în set
*/
{
        int av1;
        if ( ( gt + a[i].g) <= glim) {
                s[i] = '*';
```



```
                if ( i<N-1 )
                        incerc( i+1, gt + a[i].g, av);
                else
                        if ( av > maxv) {
                                int i;
                                maxv = av;
                                for ( i=0; i<N; i++)
                                        opts[i] = s[i];
                        }
                s[i] = ' ';

        }
        av1 = av - a[ i].v;
        if ( av1 > maxv)
                if ( i < N-1)
                        incerc( i+1, gt, av1);
                else {
                        int i;
                        maxv = av1;
                        for ( i=0; i<N; i++)
                                opts[i] = s[i];
        }
}

main()
{
        int totv = maxv = 0;
        int i, g;
        for( i=0; i < N; i++)
                totv += a[i].v;
        printf(" Greutate |");
        for( i=0; i<N; i++)
                printf("%4d", a[i].g);
        printf("\n Valoare |");
        for( i=0; i < N; i++)
                printf("%4d", a[i].v);
        printf("\n---------------------------");
        printf("---------------------------\n");
        for( g=glim=10; g <= 120; g = glim += 10) {
                printf(" %7d |", g);
                incerc( 0, 0, totv);
```

```
            for( i=0; i < N; i++)
                printf("%4c", opts[i]);
            printf("\n");
        }
        printf("--------------------------");
        printf("--------------------------\n");
}
```

Semnificația variabilelor este inclusă în program prin intermediul liniilor de comentariu.

Programul găsește seturile de obiecte cele mai valoroase care se încadrează în anumite valori de greutate (10,20,...120). Rezultatul este afișat sub forma unui tabel în care simbolul * marchează includerea obiectului în setul de greutate dată. Antetul tabelului conține greutatea și valoarea fiecărui obiect, iar prima coloană conține greutatea maximă admisă pentru setul de obiecte.

1.6. Structuri și uniuni

1.6.1. Structuri

Structura este o colecție de una sau mai multe variabile, de același tip sau de tipuri diferite ce sînt grupate sub un nume unic, pentru a putea fi tratate împreună.

Structurile ajută la organizarea datelor complexe (de regulă din programe mari) permițînd tratarea unitară a unui grup de variabile.

Elementele sau variabilele menționate într-o structură se numesc membrii ai structurii. Un membru de structură, o etichetă sau o variabilă oarecare ce nu aparține structurii pot avea același nume fără a genera conflicte, deoarece ele vor fi întotdeauna deosebite una de alta din context.

Structurile pot fi imbricate, caz în care o structură include o altă structură, și autoreferite, caz în care structura conține o referință la o structură de același tip (exemplu: elementul de listă sau de arbore).

Există patru formate de structuri. Le vom exemplifica pe o structură ce conține doi membrii: șirul *buf* de 80 de întregi și pointerul *pheap*.

Primul format are sintaxa:

```
struct bloc {
        int buf[80];
        char *pheap;
}
```


Cuvîntul cheie *struct* introduce o declarație de structură care constă dintr-o listă de declarații închise între acolade.

Cuvîntul cheie poate fi urmat opțional, de un nume numit marcaj de structură, cum este în exemplul nostru numele *bloc*. Acest marcaj denumește acest tip de structură și poate fi folosit în continuare ca prescurtare pentru declarația de structură detaliată asociată lui.

Deoarece nu există nici o variabilă de acest tip care să urmeze acoladei închise, compilatorul nu va aloca memorie pentru această structură. Ca atare, declarația de structură descrie numai un șablon sau o formă de structură. Dacă structura conține marcajul de structură, el poate fi folosit ulterior (chiar și în acea structură) pentru definirea unor alte variabile de tip structură care au același șablon cu structura marcată. De exemplu:

```
a) struct bloc var1;
b) struct lista {
        int buf;
        struct lista *link;
   }
```

Al doilea format are sintaxa:

```
struct bloc date = { listă_de_initializare };
```

caz în care structura numită *bloc* poate fi inițializată cu valorile din *listă_de_initializare*. Structura *bloc* trebuie să fie definită anterior, iar *listă_de_initializare* este opțională. În acest caz, compilatorul alocă memorie pentru variabila *date*.

Al treilea format:

```
struct bloc {
        int buf[80];
        char *pheap;
} date = { listă_de_initializare };
```

este o combinație a primelor două, iar ultimul:

```
struct {
        int buf[80];
        char *pheap;
} date = { listă_de_initializare };
```

omite marcajul de structură.

Singurele operații care se pot aplica unei structuri sînt accesul la un membru din structură și aflarea adresei lui.

Există doi operatori speciali ce lucrează cu structuri. Operatorul . selectează membrii structurii cu numele structurii. De exemplu, *date.pheap* este un pointer la caracter. Dacă avem declarația unui pointer la o structură:

```
struct bloc *ps = date;
```

operatorul -> permite referirea la un membru de structură prin adresa structurii (pointerul la ea). De exemplu, *ps->buf[5]* se evaluează la un întreg.
Se pot defini, funcție de cele prezentate, masive de structuri sau uniuni și masive de pointeri la structuri și uniuni.

1.6.2. Uniuni

O variabilă de tip uniune este o variabilă care poate conține la momente diferite, obiecte de diferite tipuri și dimensiuni. Compilatorul este cel care gestionează dimensiunea și alinierea (cadrarea).

Avantajul uniunilor constă în posibilitatea ca tipuri diferite de date să fie gestioneze aceeași zonă de memorie, fără folosirea în program a unor informații dependente de calculator.

Uniunile au sintaxă similară și utilizează aceiași operatori ca structurile. Spre deosebire de structuri, care alocă spațiu în memorie pentru fiecare membru, uniunile alocă doar spațiul necesar pentru cel mai mare membru din uniune. Spre exemplificare declarația:

```
union obiect {
        long date;
        char a[3];
} ob;
```

alocă 4 octeți. *Ob.date* este de tipul *long* și *ob.a[2]* este de tipul *char*.
Se permite declararea pointerilor la uniuni, dar nu se inițializează uniunile.
Păstrarea integrității membrilor uniunii este responsabilitatea programatorului.
Următorul program este o altă variantă pentru conversia unui număr întreg într-un număr hexazecimal.

Programul 1.16.

```
/* itoh2.c - varianta a doua la conversia */
        /* unui întreg în hexazecimal */

#include <stdio.h>

main()
{
        union {
                unsigned short num;
                unsigned char s[2];
        } val;
        char *p = "0123456789abcdef";
        printf("Introduceți un număr: ");
        scanf("%d", &val.num);
        putchar( p[val.s[1] >> 4]);
        putchar( p[val.s[1] & 15]);
        putchar( p[val.s[0] >> 4]);
        putchar( p[val.s[0] & 15]);
        putchar('\n');
}
```

Variabila *val*, de tip uniune, poate memora un întreg scurt sau un șir de doi octeți. Numărul ce se citește de la tastaură, ca un întreg scurt fără semn (2 octeți - 80286), se va memora în cîmpul *num*, iar conversia se face prin prelucrarea cîmpurilor *s[0]* și *s[1]*.

Uniunea face ca acest program să se execute mai repede decît varianta lui anterioară. În acest caz nu există bucle și se utilizează operatorii pe biți.

Programul nu este portabil, deoarece pe un calculator, diferit de 286, s-ar putea ca octeți să fie memorați în ordine inversă. Varianta portabilă o vom prezenta într-unul din capitolele viitoare.

1.7. Clase de memorie

Există patru clase de memorie, ce se utilizează împreună cu tipurile de date.

1.7.1. Clasa de memorie auto

Auto (automatic) este clasa de memorie explicită în C. Instrucțiuni ca:

auto int i, j;

sînt compilate, dar cuvîntul cheie *auto* este opțional. O variabilă ce apare într-un bloc sau în corpul unei funcții pentru care nu s-a făcut nici o declarație de clasă de memorie se consideră implicit *auto*. Declarațiile de variabile *auto* sînt locale fiecărui bloc sau funcții.

Variabilele *auto* au valori inițiale nedefinite (conțin valori reziduale) și domeniul lor de existență este blocul care le conține declarația. Înainte de utilizare variabilele *auto* trebuie inițializate. Variabilele *auto*, din interiorul funcțiilor, nu rețin valorile anterioare la următorul apel al funcției. De exemplu, următorul segment de program creează patru variabile *auto* neinițializate de patru tipuri diferite.

```
main() {
        int i;
        float f;
        char buf [80];
        struct complex {
                float imag, real;
        }
}
```

O declarație de variabile *auto* este legală oriunde un bloc este legal. Următoarele instrucțiuni:

```
if ( i>MAX) {
        char buf [80];
        ...
}
...
while ( i++<MAX) {
        char buf [80];
        ...
}
```

declară două variabile *auto*, de tip șir de caractere cu același nume (*buf*). Șirurile sînt distincte avînd fiecare zona lui separată în memorie.

În funcții diferite pot exista variabile locale cu același nume, fără ca între variabile să existe vreo legătură.

1.7.2. Clasa de memorie `static`

Clasa de memorie *static* oferă o cale de-a reține valoarea unei variabile în timpul execuției programului. Variabilele ce au această clasă de memorie pot fi: *static interne* sau *static externe*.

O variabilă de clasă *static* are valoarea zero înainte de execuția programului, fără ca ea să fie explicit inițializată.

O variabilă este de clasă *static* internă, dacă este definită în interiorul unei funcții și este precedată de cuvîntul cheie *static*. De exemplu, funcția *func()* utilizează o variabilă statică *prim* pentru a lua o anumită decizie.

```
func()
{
        static int prim = 1;
        if ( prim) {
                prim = 0;
                altăfunc();
        }
        ...
}
```

Funcția *func()* declară variabila *prim* statică și o inițializează cu valoarea *1*. La primul apel al acestei funcții se actualizează *prim* cu valoarea zero și se apelează o nouă funcție *altăfunc()*. Apelul următor al acestei funcții nu apelează funcția *altăfunc()*, deoarece variabila *prim* reține valoarea din apelul precedent. Dacă *prim* ar fi declarată *auto*, apelul funcției *altăfunc()* s-ar face la fiecare apel al funcției *func()*.

Clasa de memorie *static* nu afectează domeniul de existență al variabilelor. Ca atare variabila *prim* este necunoscută în afara funcției care o conține, deși valoarea ei se păstrează între apelurile funcției *func()*.

Variabilele statice interne permit inițializarea șirurilor și structurilor care sînt declarate în interiorul blocurilor. Instrucțiunile:

```
{
        static char buf = "exemplu";
        static struct complex {
                float real, imag;
        } val = { 2.3, 1.7};
        ...
}
```

inițializează șirul *buf* cu șirul "*exemplu*" și o variabila *val* cu numerele reale 2,3 și 1,7.

O variabilă sau funcție este numită *static* **externă**, dacă declarația ei apare în exteriorul unui bloc de funcție și este precedată de cuvîntul cheie *static*. Aceste variabile sau funcții sînt globale numai pentru fișierul sursă în care ele au fost definite, nefiind recunoscute în alte fișiere. Instrucțiunile:

```
static sem;        /* locală fișierului */
long var;          /* globală */
f()                /* globală */
{
...
}

static g()         /* locală fișierului */
{
...
}
```

definesc variabila *sem* și funcția *g()* ca static externe (tipul implicit este int), avînd domeniul de existență doar acest fișier. Alt fișier nu poate utiliza variabila *sem* și apela funcția *g()*. Un program din alt fișier poate utiliza variabila *var* și apela funcția *f()*.

Declarațiile static externe sînt așadar locale (private), iar cele fără cuvîntul cheie *static* sînt globale (publice). Într-un program mare cu multe module, trebuie selectate care dintre variabile și funcții sînt private și care publice. Este bine de-a declara toate funcțiile și variabilele nelocale statice, pentru a nu le putea accesa din alte module. Această măsură previne conflictele de nume ce pot apărea la linkeditarea modulelor program scrise de diferiți programatori.

Pentru a face lizibilă utilizarea variabilelor și funcțiilor într-un program, pentru cineva ce nu intenționează să scrie cod, se poate crea un fișier ce să echivaleze cele discutate mai sus. Acest fișier se va include în fișierele ce țin cont de aceasta echivalare. Conținutul fișierului este:

/* def.h */

```
#define PRIVATE static
#define PUBLICE
```


Exemplul anterior se modifică astfel:

```
#include "def.h"

PRIVATE sem; /* locală fișierului */
PUBLICE long var;      /* globală */
PUBLICE f()            /* globală */
{
...
}

PRIVATE g()   /* locală fișierului */
{
...
}
```

1.7.3. Clasa de memorie `register`

Limbajul C permite, prin intermediul acestei clase de memorie, utilizarea regiștrilor interni pentru memorarea variabilelor. Doar variabilele locale pot avea această clasă de memorie, iar regulile de utilizare rămîn cele de la variabilele de clasă *auto*. O variabilă de clasă de memorie *register* se declară prin specificatorul de clasă de memorie *register*. Operatorul de adresă (*&*) nu se poate aplica unei variabile *register*.

Utilizarea variabilelor *register* este dependentă de calculator și compilator. Multe compilatoare alocă regiștri doar pentru pointeri și tipurile de date *char* și *int*. Este posibil ca toate declarațiile register să fie ignorate de compilator sau să lucrăm cu ele fără a fi cerut explicit acest lucru. Consultînd documentația Turbo C rezultă că doar trei variabile de clasă *register* pot fi păstrate în regiștrii calculatorului.

Variabile de tip *register* conduc la programe mai mici și mai rapide. Operațiile aritmetice și de indexare a șirurilor din bucle, se execută mai repede în acest caz, decît în cazul utilizării variabilelor de clasă *auto* sau *static*. Variabilele de buclă, pointerii și parametrii funcțiilor pot avea clasa *register*. Utilizarea acestei clase de memorie fiind limitată, se impune o selecție. La execuție, dacă numărul declarațiilor *register* e prea mare, surplusul devine de clasă *auto*. Din acest motiv se recomandă ca declararea unui număr mare de variabile *register* să se facă în ordinea importanței lor.

Un exemplu sugestiv în acest sens este segmentul de cod:

```
register int j;
register int i;
int b[IMAX][JMAX];
...
for ( i=0; i<IMAX; i++)
        for ( j=0; j<JMAX; j++)
                ...b[i][j]...
```

A doua buclă *for* se execută mai des decît prima, motiv pentru care variabila *j* este mai importantă ca variabila *i* şi ca atare este declarată prima.

1.7.4. Clasa de memorie `extern`

Clasa de memorie *extern* permite unui modul program utilizarea variabilelor şi funcţiilor din alte module program. Toate declaraţiile care nu conţin cuvîntul cheie *static* sînt accesibile din alte module. Fişierul care utilizează variabile şi funcţii din alt modul trebuie să conţină declaraţia acestora precedată de cuvîntul cheie *extern*. Variabilele externe sînt variabile cu caracter global. Ele se definesc în afara oricărei funcţii şi pot fi apelate prin nume din oricare funcţie a unui program.

Funcţiile sînt considerate în general variabile externe în afară de cazul cînd se specifică altfel.

Variabilele externe se folosesc deseori pentru a comunica date între funcţii. De exemplu, fie două fişiere **`mod1.c`** şi **`mod2.c`**. Primul apelează o funcţie definită în al doilea, iar al doilea la rîndul lui, utilizează date declarate în primul. Codul primului fişier este:

```
/* mod1.c */
#include <stdio.h>
#include "global.h"
int nrart;
double şir[ 100];
struct orice s;
double f();
main()
{
        ...
        sir[ 7]=f();
        ...
}
```


Se definesc întregul *nrart*, şirul *şir* şi variabila *s* de tipul *orice*. Tipul *orice* este definit în fişierul **`global.h`**. Toate aceste variabile pot fi utilizate în **`mod2.c`**, deoarece nici una nu este declarată statică. Funcţia *main()* apelează funcţia *`f()`*, care nu este definită în **`mod1.c`**. Codul pentru fişierul al doilea este:

```
/* mod2.c */

#include <stdio.h>
#include "global.h";

extern int nrart;
extern double şir[];

double f()
{
        extern struct orice s;
        ...
}
```

Fişerul **`mod2.c`** foloseşte cuvîntul cheie *extern*, pentru a putea utiliza variabilele *nrart*, *şir*, şi *s* din fişierul **`mod1.c`**. Dimensiunea şirului *şir* este opţională în declaraţie.

Funcţiile fiind implicit externe declaraţia:

```
extern double f();
```

în **`mod1.c`** este redundantă.

1.8. Directive preprocesor şi macrouri

Preprocesorul este separat de compilator şi are propriile reguli de sintaxă. Preprocesorul:
- tratează definiţilie de constante simbolice,
- oferă un mecanism pentru definirea de macrouri,
- tratează directive pentru compilarea condiţionată,
- include eventuale fişiere
- realizează substituţii textuale (macroexpandări)

1.8.1. Directive preprocesor

Orice directivă preprocesor începe cu caracterul #. Caracterul \ la sfîrşit de linie marchează continuarea directivei pe linia următoare. O directivă este permisă oriunde în codul sursă (este posibilă şi imbricarea lor).

Directivele:

```
#include "fişier"
#include <fişier>
```

determină includerea liniilor fişierului *fişier*, în programul care le conţine. Diferenţa dintre cele două directive constă în faptul că prima determină o căutare suplimentară în directorul curent pe lîngă căutarea în directorul ce conţine biblioteca standard.

Directivele:

```
#define NUME text
#undef NUME
```

activează şi dezactivează substituţia textuală. Ca atare, se substituie orice apariţie a lui *NUME*, ce nu se află între ghilimele ("*NUME*"), cu textul *text*.

Directivele:

```
#if expresie_constantă
...
#endif

if expresie_constantă
...
else
...
#endif

if expresie_constantă
...
elif expresie_constantă
...
#endif
```


determină compilarea condiţionată. Dacă *expresie_constantă* este evaluată la o valoare diferită de zero se compilează liniile dintre *#if* şi *#endif*. În caz contrar preprocesorul elimină liniile din codul sursă, iar compilatorul nu le compilează.

Directivele:

```
#ifdef NUME
...
#endif


#ifndef NUME
...
#endif
```

determină compilarea liniilor ..., dacă *NUME* a fost definit respectiv nu fost definit.

1.8.2. Macrouri

Macrourile utilizează directiva *#define* şi au sintaxa:

```
#define NUME( arg1,arg2,...,argn) text
```

Preprocesorul substituie textual în codul sursă toate apariţiile argumentelor *arg1*, *arg2*, ... în textul *text*. De regulă macrourile conţin operatorul ternar (*? :*), dar pot conţine blocuri şi instrucţiuni de control.

Un exemplu este calculul numărului de elemente pentru un şir de tip necunoscut:

```
#define NELEM( A) (sizeof( A)/ sizeof A[0])
```

Următorul grup de macrouri permite iniţializarea şi testarea valorii anumitor biţi.
Macrourile utilizează operatori pe biţi şi folosesc parantezele pentru separarea textului şi a parametrilor. Acest lucru permite apariţia lor în orice expresie C.

```
#define SET(X,N) ((X) |= ( 1 << (N)))
#define RESET(X,N) ((X) &= ~( 1 << (N)))
#define TEST(X,N) ((X) & ( 1 << (N)))
```

Presupunem că variabila *a* conține o valoare întregă. *SET(a,5)* poziționează bitul *5* la *1*, *RESET(a,5)* poziționează același bit la *0*, iar expresia:

```
if ( TEST(a,4)) {
        ...
}
```

execută blocul instrucțiunii *if*, dacă bitul *4* a lui *a* este *1*. Un ultim exemplu, ce realizează interschimbul a două variabile de tip *char*, *short*, *int*, sau *long* este:

```
#define SWAP(A,B) { \
        long t;
        (t) = (A); \
        (A) = (B); \
        (B) = (t); \
}
```

Macrourile se execută mai repede decît funcțiile echivalente lor, deoarece compilatorul nu necesită punerea parametrilor pe stivă și returnarea valorilor. Macrourile, pe de altă parte, măresc dimensiunea codului și au unele restricții de utilizare. Dintre ele:

- un macrou nu se poate apela dintr-un alt limbaj de programare,
- nu există pointeri la macrouri (pe cînd la funcții există)
- nu toate funcțiile pot fi convertite în macrouri. Macrourile pot genera efecte laterale, dacă utilizează argumente cu operatorii ++ și --. De exemplu instrucțiunile:

```
FILE *fp;
char *p;
...
putc(*p++, fp);
```

utilizează macroul *putc* (definit în fișierul **stdio.h**) pentru a trimite un caracter în fișierul referit de pointerul *fp*. Instrucțiunea se execută incorect, deoarece pointerul *p* poate fi incrementat mai mult decît o dată în momentul substituției textuale (macroexpandării). Pentru evitarea acestei probleme, pointerul *p* trebuie incrementat în afara apelului sau utilizată funcția *fputc()*.

1.9. Castul

Operatorul de forțare a tipului este numit *cast* și are formatul:

```
(tip) expresie;
```

Castul convertește rezultatul evaluarii expresiei *expresie* la tipul *tip* specificat între paranteze. Valoarea variabilelor din expresie nu este afectată. Castul se utilizează cu precădere în funcții pentru conversia tipului parametrilor și-a tipului valorii returnate. Ei afectează conversiile aritmetice și ajută la construcția unui cod portabil.

Dacă variabila *i* și *j* sînt întregi, instrucțiunea:

```
pow((double) i, (double) j);
```

apelează funcția *pow()*, din biblioteca matematică, ce ridică pe *i* la puterea *j*. Castul convertește tipul variabilelor *i* și *j* la tipul double fără a afecta valoarea lor.

Instrucțiunile:

```
struct bloc *bptr;
...
bptr=(struct bloc *)malloc(sizeof(struct bloc));
```

alocă spațiu în memoria heap pentru o structură de tipul *bloc* și atribuie adresa de început a acestei zone pointerului *bptr*. Castul este necesar pentru a realiza conversia de la pointer la void la pointer la structura *bloc*.

Presupunînd că *val* este inițializată cu o valoare flotantă și *num* este inițializată cu o valoare întreagă expresia:

```
(int) val/num;
```

generează rezultatul ca întreg, deoarece castul convertește rezultatul împărțirii flotante la întreg.

1.10. Probleme propuse pentru rezolvare

1. Completați programul **rot.c** cu instrucțiuni care verifică dacă numărul deplasărilor spre dreapta sau stînga nu este mai mare decît dimensiunea în biți a numărului.

2. Modificați funcțiile *rol* și *ror*, astfel încît ele să modifice numărul întreg primit ca argument (se transmite ca argument pointerul la număr).

3. Pentru a genera numere aleatoare se pot utiliza operatorii pe biți. Dacă *i* este o variabilă de tip întreg expresia:

```
i=(i& 0x8000) ? (i<<1) ^0x100b : (i<<1);
```

actualizează variabila *i* cu o nouă valoare funcție de valoarea anterioară a sa.

Să se scrie o funcție *rnd()*, care utilizînd această expresie să genereze numere aleatoare. Pentru ca funcția să fie corectă nu uitați să inițializați variabila *i* cu o valoare diferită de zero.

4. Să se scrie o funcție *nrzero()*, care primind ca argument un număr întreg, determină numărul de biți egali cu zero din reprezentarea internă a numărului.

5. Să se scrie un program care inversează un șir de caractere. Șirul se citește din linia de comandă, iar prelucrarea sa se face utilizînd argumentul *argv*.

6. Modificați funcția *swap()* pentru a interschimba două valori flotante fără a utiliza o variabilă temporară. Ce restricție ar avea această funcție și cît de portabil este codul ?.

7. Modificați macroul *SWAP* astfel încît el să nu utilizeze o variabilă temporară. Care este dezavantajul ?.

8. Compilați și executați programul:

```
/* sizeof.c - operatorul sizeof */

main()
{
        long buf[50];
        double *p;
        struct orice {
                char buff[80];
        } s;
        printf("%d\n", sizeof(int));
        printf("%d\n", sizeof(double));
        printf("%d\n", sizeof(double *));
        printf("%d\n", sizeof(struct orice));

        printf("%d\n", sizeof buf);
        printf("%d\n", sizeof buf[2]);
        printf("%d\n", sizeof buf+1);
        printf("%d\n", sizeof s.buff);
}
```


Puteți explica rezultatul execuției ?.

9. Programul următor folosește o stivă și două funcții *push()* și *pop()* pentru a inversa un șir de caractere.

```
/* stivă.c */

#include <stdio.h>

#define NSTACK 20
char stack[ NSTACK], *pstk=stack;

main()
{
        char *p="emit are";
        int i;
        while ( (i=*p++) != '\0')
                push(i);
        while (( i=pop()) != EOF)
                putchar(i);
        putchar('\n');
}

push( char d)
{
        if ( pstk >= stack+NSTACK)
                exit(1);
        *pstk++=d;
}

pop()
{
        if ( pstk <= stack)
                return EOF;
        return *--pstk;
}
```

Să se rescrie funcțiile `push()` și `pop()` ca macrouri.

10. Să se includă în programul **stivă.c** directive preprocesor ce să permită compilarea programului cînd cu macrouri cînd cu funcții. Se utilizează opțiunea -*D* din linia de comandă.

11. Să se scrie un program ce realizează operațiile de bază ale unui calculator simplu. Operațiile sînt: adunare (+), scădere (-), înmulțire (*) și împărțire (/). Operanzii și operatorii se citesc din linia de comandă.

12. Să se rescrie programul **caluț.c** asfel încît să se găsească zece soluții ale problemei.

13. Să se rezolve folosind revenirea prin nedeterminare problema celor opt dame. În cîte moduri se pot amplasa pe o tablă de șah opt dame fără ca ele să se "atace" între ele.

14. Să se calculeze factorialul unui număr citit din linia de comandă. Numărul este considerat a fi mai mic sau egal cu 76.

15. Completați programul **rec3.c** cu instrucțiunile și funcțiile lipsă. Programul este portabil ?.

16. Explicați programul **viewhanoi.c**. Separați și explicați acțiunea fiecărei funcții, stabiliți condiția de terminare pentru funcția recursivă și analizați structura programului.

```
/* viewhanoi.c - problema turnurilor din Hanoi */

/* vizualizarea conținutului turnurilor */

#define N 10
#include <stdio.h>

/*      declarații de funcții */

void hanoi(int *, int *, int *, int);
void mutare(int *, int *);
int pop(int *);
void push(int *, int);
void print(void);
void prints(char, int *);


int sa[N] = { 9,9,8,7,6,5,4,3,2,1 };
int sb[N];
int sc[N];
```



```
main( void)
{
        int m;
        sa[0]=m=3;
        print();
        hanoi(sa,sb,sc,m);
}

void hanoi( int *a,int *b,int *c,int n )
{
        if( n==1 )
                mutare(a,b);
        else {
                hanoi(a,c,b,n-1);
                mutare(a,b);
                hanoi(c,b,a,n-1);
        }
}

void mutare( int a[], int b[] )
{
        push(b, pop(a));
        print();
}

int pop( int s[] )
{
        if( s[0]==0 )
                printf("pop: stiva vida\n");
        else
                return(s[(s[0]--)]);
}

void push ( int s[], int e )
{
        s[++s[0]]=e;
}
```

```
void print( void )
{
        prints('A', sa);
        prints('B', sb);
        prints('C', sc);
        putchar('\n');
}

void prints( char c, int *s)
{
        int i;
        printf("%c: ", c);
        for( i=0; i < s[0]; i++ )
                printf("%2d", s[i+1]);
        putchar('\n');
}
```

Ce returnează funcția pop ( ) prin expresia: *s[(s[0]--)]* ?
Trasați și executați programul.

2. Mediul de execuție

Mediul de execuție este o caracteristică specifică limbajului C. În majoritatea limbajelor, declarația unei variabile conține **numele simbolic (identificatorul)**, **clasa** (șir, structură, reuniune etc) și **tipul** variabilei. Avînd în vedere structura unui program C, locul plasării unei declarații este esențial pentru domeniul de existență al variabilei. Compilatorul nu decide, programatorul însă da. Acest lucru constituie un avantaj, deoarece programatorul poate exercita un control mai mare in timpul execuției programului. Neajunsul acestei facilități rezidă în necesitatea unei viziuni clare a programatorului asupra modului de execuție a programului.

Se analizează în cele ce urmează cîteva aspecte esențiale pentru mediul de execuție. Dintre ele:
- separarea unui program în zone program și efectul pe care clasele de memorie îl au asupra acestor zone.
- caracterizarea variabilelor din program și estimarea necesarului de memorie la execuție.
- caracterizarea variabilelor din program în contextul mediului de execuție (alocare memorie, specificare clasă de memorie).

Cu toate că mediul de execuție e dependent de hardul calculatorului, înțelegerea conceptului oferă programatorului cheia scrierii unor programe care necesită memorie mică și viteză în execuție.

2.1. Zone program

Sub orice sistem de operare, fie el UNIX, OS/2 sau DOS, la execuția unui program, sistemul de operare va aloca la început o zonă de memorie pentru program. Codul executabil si structurile de sistem asociate formează așa numitul proces (din UNIX sau OS/2), iar sistemul de operare copiază programul executabil de pe disc în memorie. Bineînțeles, această copiere nu se reduce doar la instrucțiunile executabile. Sistemul de operare va aloca memorie și pentru alte părți ale programului. O separare în zone a mediului de execuție ar putea fi: zona text, zona stivă, zona de date și zona heap.

Zona text este acea zona rezervată din memorie ce conține instrucțiunile executabile ale programului. Zona este protejată la scriere și se poate accesa prin pointeri la funcții.

Zona de date este zona rezervată memorării variabilelor statice (interne sau externe) și variabilelor globale din program. Zona conține atît variabile inițializate, cît și neinițializate, motiv pentru care zona este împărțită în două.

Zona stivă conține datele (organizate în cadre stivă) necesare mecanismului prin care compilatorul implementează apelul de funcții și variabilele *auto* ale programului.

Zona heap este partea din memorie unde se fac alocările dinamice (din timpul execuției). Gestionarea acestei zone este sarcina programatorului.

Zona text și zona de date sînt fixe ca dimensiune, întucît lungimea programului și a datelor este cunoscută în momentul execuției. Zona stivă și zona heap sînt dinamice, deoarece ele variază în momentul execuției. Programul utilizează aceste zone la execuție, iar alocarea de memorie se bazează pe structura programului, pe operațiile de intrare/ieșire și pe apelul de funcții. În momentul încărcării, lungimile lor sînt necunoscute. Zona stivă este dependentă de zona heap și invers. Creșterea uneia duce la scăderea celeilalte. Acest lucru este valabil atîta timp cît zona stivă si zona heap partajează aceași zonă de memorie.

Dispunerea acestor zone în memorie nu interesează, esențial este necesarul de memorie pentru program. Acest lucru ajută în depanarea erorilor la execuție (cum ar fi depășirea stivei sau depășirea zonei heap).

Frecvent zona de stivă si heap nu partajează aceeași zonă de memorie, salvarea spațiului stivă putînd să nu afecteze spațiul necesar zonei heap. Spațiul necesar pentru memorarea datelor unui program afectează zonele text, stivă sau heap. Aceste detalii de implementare depind de arhitectura calculatorului, de sistemul de operare și de opțiunile liniei de comandă la compilare și încărcare. Esențial este a stabili care părți din program sînt dinamice și care sînt fixe.

2.2. Zona text

Zona text este porțiunea din memorie rezervată instrucțiunilor executabile dintr-un program (și nu caracterele *ASCII* ce formează fișierul sursă). În general, această zonă este considerată protejată la scriere. Compilatoarele C au în general opțiuni de compilare ce plasează zona text și zona datelor în locuri separate. În acest caz ele sînt numite zone **pure**. Ele pot însă partaja aceași zonă de memorie, caz în care se numesc zone **impure**. Zona text are dimensiune fixă, determinată în momentul compilării și linkeditării. Singura cale prin care un program accesează zona text este prin pointeri la funcții.

2.2.1. Pointeri la funcții

La execuția unui program sînt parcurse instrucțiunile din zona text. Programele C nu pot accesa zona text decît utilizînd pointeri la funcții. În acest caz, pointerul conține o adresă a unei funcții din zona text, iar programul trebuie să parcurgă adresele din zona text.

Pentru a declara un pointer la o funcție ce returnează o valoare întregă, folosim declarația:

```
int (*p)();
```

Un pointer la o funcție trebuie inițializat cu adresa funcției. Acest lucru se face în două etape:
a) se specifică compilatorului numele funcției;
b) se atribuie adresa funcției pointerului.
De exemplu liniile:

```
int func();
p = func;
```

Prima linie informează compilatorul că *func()* este o funcție care returnează o valoare întregă. La acest pas nu e necesară specificarea parametrilor, fiind suficientă doar specificarea tipului lor (a se vedea capitolul 1). Esențială e cunoașterea numelui funcției și a tipului de dată returnat de ea.
A doua linie inițializează pointerul *p* cu adresa funcției *func()*. Numele funcției nu este urmat de paranteze, deoarece nu este apel de funcție. Compilatorul interpretează numele funcției ca o adresă, deci operatorul *&* nu este necesar.

O altă modalitate de inițializare a unui pointer la o funcție constă într-o declarație ce combină etapele prezentate anterior:

```
int func(), (*p)() = func;
```

Apelul funcției se realizează astfel:

```
rezultat = (*p)(arg1, arg2, ...);
```

ce invocă funcția *func()* cu argumentele sale (dacă ele sînt necesare) și memorează valoarea returnată în variabila *rezultat* (ce trebuie să fie declarată de tip întreg). Toate parantezele sînt obligatorii în acest caz. Se consideră exemplele:

Programul 2.1.

```
/* pow.c - ridicarea la putere */

#include <stdio.h>
#include <math.h>

main( int argc, char *argv[])
{
        double baza, exponent;
        double ( *mtab[ 10])();
        if ( argc != 3) {
                printf("Utilizare: %s baza exponent\n",argv[0]);
                exit(1);
        }
        baza = atof( argv[1]);
        exponent = atof( argv[2]);
        mtab[0] = pow;
        printf(" %g\n", (*mtab[0])(baza,exponent));
}
```

S-a declarat şirul *mtab* de zece pointeri la funcţii care returnează *double*. Primul element din şir este iniţializat cu adresa funcţiei *pow()*, iar restul elementelor sînt neiniţializate. Apelul funcţiei se realizează prin instrucţiunea:

```
( *mtab[0])( baza, exponent);
```

Parametrii actuali *baza* şi *exponent* au tipul *double*, în concordanţă cu tipul returnat de funcţia *atof()*. Programul necesită două argumente în linia de comandă şi includerea fişierului **math.h**, ce conţine declaraţia funcţiei *atof()*. Formatul "*%g*" tipăreşte partea zecimală a rezultatului, dacă ea există.

În exemplul următor se creează un tabel cu funcţii trigonometrice din biblioteca matematică. Utilizăm o structură cu doi pointeri: unul la numele funcţiei şi altul la funcţia propriu-zisă.

Programul 2.2.

```
/* trig.c - funcţii trigonometrice */

#include <math.h>
#include <stdio.h>
```



```
main()
{
        struct ftrig {
                char *numef;
                double (*ptrf)();
        };

        static struct ftrig tabl[10]={
                "acos", acos,
                "asin", asin,
                "atan", atan,
                "atan2", atan2,
                "cos", cos,
                "cosh", cosh,
                "sin", sin,
                "sinh", sinh,
                "tan", tan,
                "tanh", tanh
        };

        int i;
        for (i=0; i<10; i++)
                printf("%04x\t%s\n", tabl[i].ptrf, tabl[i].numef);
}
```

În funcţia *main()* s-a declarat structura *ftrig* ce conţine un pointer la caracter şi un pointer la o funcţie ce returnează o valoare de tip *double*. Se declară şirul *tabl* de tipul structurii *ftrig* şi se iniţializează elementele sale cu funcţii trigonometrice din biblioteca C. În final, se parcurge şirul şi se afişează adresa şi numele fiecărei funcţii. Adresele sînt afişate în format hexazecimal şi reprezintă adrese din zona text. Rezultatul execuţiei este:

```
04c6    acos
04dc    asin
04f1    atan
04fe    atan2
05a9    cos
05f5    cosh
0fd7    sin
1023    sinh
10c4    tan
1113    tanh
```

Să considerăm un program ce încearcă să scrie ceva în zona text. Limbajul C permiţînd pointeri la diverse lucruri, este foarte posibilă o utilizare incorectă a lor. Pentru demonstraţie, fie programul:

Programul 2.3.

```
/* modif.c - modificarea propriului cod */

#include <stdio.h>

main()
{
        void func();
        strcpy(( char *) func, "reziduu");
        printf("adresa funcţiei = %04x\n", func);
        func();
        printf("<- Am revenit din funcţie\n");
}

void func()
{
        printf("-> Sînt în funcţie \n");
}
```

Se apelează funcţia *strcpy()* cu două argumente. Primul argument este un pointer la funcţia *func()*, care este convertit în pointer la caracter prin cast. Al doilea argument este un pointer la şirul *"reziduu"*. Funcţia *strcpy()* scrie şirul *"reziduu"* în zona text, adică la adresa funcţiei *func()*.

Acest program apare ca lucrînd corect deşi el nu are această calitate. Iniţial afişăm adresa funcţiei, iar apoi apelăm funcţia.

Se constată că sistemul de operare DOS nu afişează nici un mesaj de avertizare, deşi zona text a programului a fost modificată la execuţie.

2.3. Zona stivă

Zona stivă (stiva) este dinamică şi dimensiunea ei depinde de modul de execuţie al programului. Spre deosebire de zona text, care are dimensiune constantă, stiva poate creşte sau descreşte. Un program poate să nu utilizeze aceaşi zonă stivă la execuţii succesive. Apelurile de funcţii, structura programului şi declaraţiile de variabile definesc modul în care compilatorul citeşte sau scrie

din şi în zona stivă. Utilitatea stivei este în aşa numitele *cadre stivă*, care constituie mecanismul utilizat de compilator la implementarea apelurilor de funcţii. De exemplu, la transmiterea unui parametru unei funcţii, cadrul stivă furnizează datele necesare funcţiei. Este esenţială cunoaşterea modului în care se utilizează stiva din cel puţin două motive. Primul permite ca prin utilizarea variabilelor de clasă de memorie *auto* să se facă economie de memorie şi să se estimeze necesarul de stivă. Al doilea motiv oferă o bază teoretică pentru depanarea programelor.

Pentru stivă esenţiale sînt:

- modul în care sistemul alocă şi iniţializează variabilele *auto* şi
- modul de gestionare al stivei în funcţii, blocuri şi bucle.

Stiva este o structură de date, care memorează informaţiile într-o manieră caracteristică. În C (Turbo C) stiva este utilizată cel mai adesea pentru memorarea adreselor şi anumitor date. Sistemul introduce sau elimină (încarcă sau descarcă) datele în sau din stivă, în maniera ultimul intrat primul ieşit (**LIFO**). C-ul memorează în stivă:

- variabile auto,
- parametrii funcţiilor,
- adresele de revenire pentru funcţii,
- variabile temporare din timpul evaluării expresiilor.

O astfel de grupare o numim **cadru stivă**. Cadrele stivă sînt distincte ori de cîte ori sistemul apelează o funcţie. Ca atare, stiva programului este compusă dintr-o succesiune de cadre ce se succed pe măsura execuţiei programului.

Compilatorul poate transmite parametrii funcţiilor şi prin intermediul regiştrilor interni ai calculatorului. Astfel se cîştigă viteză la execuţie.

Se consideră două programe ce utilizează cadrele stivă la apelul de funcţii. Primul program apelează funcţia *func()* cu adresele a două variabile întregi.

Funcţia *func()* utilizează în blocul ei două variabile întregi *m* şi *n*.

Programul 2.4.

```
main()
{
        int a=3, b=4;
        func(&a, &b);
}
```

```
func( int *pa,int * pb)
{
        int m=5, n=7;
}
```

Compilatorul memorează în cadrul stivă argumentele funcţiei, variabilele locale şi adresa de revenire în funcţia apelantă. În *figura 2.1* este ilustrată harta memoriei stivă imediat după apelul funcţiei *func()*. Programul este executat pe un calculator cu o arhitectură de 16 biţi. În figură adresele de memorie apar în stînga cadrului stivă, iar numele variabilelor programului şi adresele de cadre stivă în dreapta cadrului.

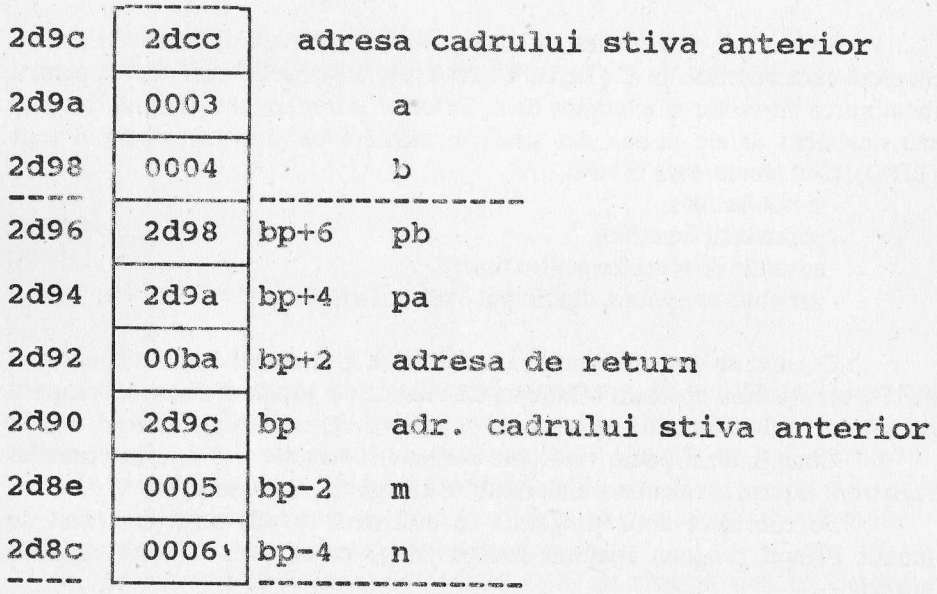


Fig.2.1. Cadre stivă

Compilatorul utilizează un pointer *bp* (base pointer) ce referă cadrul stivă curent. Imediat după apelul *func()*, *bp* referă adresa *2d90*. Adresa imediat superioară conţine adresa de revenire (*00ba*). Urmează cele două argumente ale funcţiei, adresele a două variabile întregi *pa* şi *pb*, adică adresa variabilelor *a* şi *b* din programul principal. Din figură se observă posibilitatea accesului la parametrii funcţiei printr-un deplasament pozitiv relativ la *bp*.

Variabilele locale *m* şi *n* ale funcţiei *func()*, au adrese mai mici faţă de *bp*. Deci, ele pot fi accesate cu un deplasament negativ relativ la *bp*.

Cadrul stivă curent conţine adresa cadrului stivă anterior (*2d9c*). Aceasta este adresa cadrului stivă pentru funcţia *main()*. Adresele *2d9a* şi *2d98* conţin variabilele locale ale funcţiei *main()*. Organizarea este similară variabilelor locale *m* şi *n* din interiorul cadrului stivă curent. Linii punctate delimitează cadrul stivă curent.

În momentul în care se apelează o funcţie din interiorul unui program sistemul creează un cadru stivă şi actualizează *bp* cu adresa cadrului stivă. Se generează apoi codul pentru încărcare pe stivă a argumentelor şi a adresei de revenire. Compilatorul generează codul în limbaj de asamblare utilizînd pointerul *bp* pentru a referi argumentele funcţiei (cu deplasament pozitiv) şi variabile auto (cu deplasament negativ). Dacă funcţia apelează o altă funcţie, sistemul va crea un nou cadru stivă, iar procesul anterior se repetă. Cînd se revine din funcţie, sistemul descarcă adresa de revenire din cadrul stivă curent, iar cadrul stivă anterior devine curent. În exemplul considerat *bp* este actualizat cu adresa *2d9c* în momentul revenirii din funcţia *func()*.

De exemplu, fie o funcţie care are ca argumente variabile şi nu adresele lor. Codul este:

Programul 2.5.

```
main()
{
        int a=3, b=4;
        func(a, b);
}

func( int c,int d)
{
        int m=5, n=7;
}
```

Cadrul stivă este diferit doar pentru variabilele *c* şi *d* (adresele *2d94* şi *2d96*) ce conţin acum copii ale valorilor variabilelor *a* şi *b*. Valorile din variabilele *a* şi *b* sînt izolate de *func()*. Funcţia utilizează copia memorată în cadrul stivă curent şi, ca atare, nu poate modifica sau referi valoarea variabilei *a* sau *b*. Aceasta e diferenţa esenţială între transferul prin adresă şi transferul prin valoare.

Compilatorul din TC introduce implicit în stivă parametrii funcţiei de la dreapta la stînga. Ca atare, în momentul execuţiei funcţiei, primul parametru al ei

se găseşte la adresa *bp+4* (vîrful stivei). Ordinea de transmitere a parametrilor funcţiei prin cadre stivă poate fi şi inversă.

Acest mecanism permite explicarea modului de implementare al funcţiilor *scanf()* şi *printf()*. Fie codul:

```
int val;
char buf[80];
printf("%d %s", val, buf);
```

Dacă adresa şirului *buf* este primul argument care se introduce în stivă rezultă că şirul "*%d %s*" (formatul de scriere) se află în vîrful stivei. Aşadar funcţia *printf()* utilizează această informaţie pentru a determina numărul de argumente şi pentru a le interpreta (adresă sau valoare). În capitolul 4 se indică metoda de scriere a unei funcţii cu număr variabil de argumente.

În exemplul prezentat se transmit funcţiei pointeri la tipul întreg. Nu apare nici o diferenţă dacă aceşti pointeri referă structuri, deoarece pointerul este reprezentat pe doi octeţi. Dacă transmitem o structură ca argument unei funcţii, sistemul duplică structura şi rezultă un cadru stivă mai larg. Creşte spaţiul stivă, iar compilatorul generează cod în limbaj de asamblare pentru a copia în cadrul stivă toţi membrii structurii. Toate acestea duc la încetinirea execuţiei programului.

Această problemă nu apare la şiruri, deoarece în acest caz se transmite adresa primului element al şirului.
Dacă se doreşte însă transmiterea unui şir explicit, ca parametru într-o funcţie, se utilizează tehnica prezentată în capitolul 4.

2.3.1. Alocarea stivei

Tipul unei variabile `auto` determină numărul de octeţi pe care compilatorul îi alocă acesteia pe stivă. Dacă pentru tipul `char` se alocă un octet, pentru o variabilă de tip şir sau structură se va aloca un număr de octeţi dependent de lungimea şirului respectiv a structurii.

Compilatorul trebuie să aloce memorie pentru variabilele *auto* la apelul unei funcţii. Cunoscînd acest detaliu putem estima cît spaţiu stivă necesită o funcţie. De exemplu, fie un program ce apelează două funcţii *f1()* şi *f2()*. Fiecare funcţie conţine declaraţia unei variabile şir, de clasă *auto*.

Programul 2.6.

```
main()
{
        char buf1[100];
        f1();
}
f1()
{
        char buf2[500];
        f2();
}
f2()
{
        char buf3[500];
}
```

Programul necesită minim 1100 de octeţi (excluzînd octeţii pentru apelul funcţiei), adică suma dimensiunii şirurilor. Funcţia *f1()* utilizează un cadru stivă de *500* de octeţi, care nu poate fi eliberat pînă cînd nu se revine din funcţia *f2()*, deoarece *f1()* conţine apelul funcţiei *f2()*. Analog funcţia *main()* nu poate elibera cei *100* de octeţi.

Dacă programul se modifică astfel:

```
main()
{
        char buf1[100];
        f1();
        f2();
}
```

necesarul de memorie va scădea, pe aceleaşi considerente, la *600* octeţi, funcţiile *f1()* şi *f2()* putînd utiliza succesiv aceeaşi zonă de memorie.

În C sînt permise declaraţiile de variabile *auto* în interiorul blocurilor. O astfel de declaraţie afectează domeniul de existentă al acelei variabile şi nu spaţiul stivă. Fie segmentul de program:

```
void funct( int mod)
{
        if ( mod == 1) {
                char buf[100];
        } ...
...
}
```

O greşeală frecventă în acest caz ar fi afirmaţia că alocarea de spaţiu stivă pentru variabila *buf* se face numai dacă variabila *mod* are valoarea 1. Compilatorul va aloca memorie stivă pentru toate variabilele *auto*, indiferent dacă sîntem sau nu în domeniul lor de existentă. Acest lucru serveşte sistemului de operare pentru detectarea depăşirii stivei, înainte ca programul să apeleze o funcţie.

2.3.2. Iniţializarea variabilelor auto

Ne-am aştepta ca iniţializarea variabilelor *auto* să se facă implicit la deciarare (ca în Pascal), dar ele au valoare nedefinită la declarare. Fie programul:

```
main()
{
        int i;
}
```

Codul în limbaj de asamblare generat pentru acest program relevă existenţa variabilei *i* într-un cadru stivă. Nu va exista nici o linie care să iniţializeze variabila *i*. Dacă modificăm programul:

```
main()
{
        int i=7;
}
```

se alocă spaţiu pentru *i*, în cadrul stivă, şi se încarcă *7* în locaţia corespunzătoare ei. Ca atare, compilatorul reduce din start pasul de iniţializare, dacă acesta nu este cerut explicit.

Pentru programatorii începători greşeala următoare este frecventă. Funcţia *test()* verifică dacă valoarea *val* apare printre elementele unui şir. Codul ei este:

```
int test( int * buf, int nelem, int val)
{
        int i, nc;
        for ( i=0; i<nelem; i++)
                if ( buf[ i] == val)
                        nc++;
        return nc;
}
```


Funcţia *test()* are o eroare la execuţie. Variabila *nc* este neiniţializată. Programul ar putea să se execute corect, doar dacă, întîmplător, valoarea din stivă corespunzătoare variabilei *nc* este zero.

Pentru variabile *auto* din bucle considerăm programul de mai jos:

Programul 2.7.

```
#include <stdio.h>

main()
{
        int i;
        for ( i=0; i<3; i++) {
                int j=10;
                printf("i= %d \t j= %d\n", i, j);
                j++;
        }
}
```

Execuţia programului afişează:

i= 0 j= 10
i= 1 j= 10
i= 2 j= 10

Dacă variabila *j* este memorată în cadrul stivă la adresa *bp-4* instrucţiunea C de iniţializare a variabilei se substituie cu: *MOV [bp-4], 10*. Această instrucţiune introduce valoarea *10* în cadrul stivă la fiecare intrare în buclă. Se recomandă să nu se facă iniţializarea de variabile *auto* în bucle, dacă ele pot fi declarate statice şi iniţializate în afara buclelor.

Se consideră iniţializarea şirurilor şi structurilor *auto*. Instrucţiunea:

```
double real = 25.27;
```

determină compilatorul să aloce spaţiu stivă şi să genereze cod pentru încărcarea constantei reale *25,27* în locaţia corespunzătoare variabilei *real*. Pentru un şir de tipul *double*:

```
double şirreal[4] = { 1.2, 3.4, -1.2, 10.};
```

compilatorul alocă spațiu, rămînînd ca inițializarea elementelor șirului să se facă doar la execuție. Compilatorul va genera cod pentru a copia toate constantele în spațiul stivă. În acest caz, șirul avînd doar patru elemente, inițializarea lor necesită cod în asamblare relativ scurt. Dacă însă numărul lor este mare, apare o întîrziere semnificativă la execuție. Se recomandă utilizarea variabilelor statice, deoarece inițializarea se va face înainte de execuția programului.

```
static double șirreal[4]={1.2,3.4,-1.2,10.};
```

Rămîne astfel la latitudinea programatorului decizia alegerii clasei de memorie adecvate pentru variabilele programului, funcție de cerințele problemei.

2.3.3. Funcții recursive

La apelul recursiv al unei funcții, sistemul va crea pe stivă un nou set de variabile *auto*. Este evidentă, așadar, dependența directă dintre creșterea stivei și numărul de apeluri recursive ale funcției. Utilizarea apelului recursiv conferă programelor C o anumită eleganță, însă necesită o atenție deosebită la necesarul de stivă. Dacă apelul recursiv este prea frecvent, s-ar putea depăși capacitatea stivei (*stack overflow*).

Să considerăm ca exemplu programul ce convertește un număr întreg în hexazecimal, prin utilizarea recursivă a funcției *itoh()*. Vizualizăm la fiecare apel conținutul stivei. Programul are ca argument, în linia de comandă, întregul ce urmează a fi convertit. Algoritmul de conversie utilizează restul împărțirii la 16. Împărțim numărul la 16 atîta timp cît rezultatul împărțirii este diferit de zero și afișăm restul acelor împărțiri:

```
putchar( hex[n % 16];
```

n % 16 realizează *n* modulo 16 și servește ca index în șirul *hexa*.

Programul 2.8.

```
/* itoh3.c - conversie recursivă de la întreg la hexazecimal */
#include <stdio.h>
int nivrec;
main( int argc, char *argv[])
{
        unsigned int num;
/*      freopen( "salv", "w", stderr); */
        if ( argc == 1) {
                fprintf( stderr, "Apel: %s număr\n", argv[0]);
                exit(1);
        }
        num=atoi( argv[1]);
        itoh( num);
        putchar('\n');
}

itoh( unsigned int n)
{
        int digit;
        char *hex = "0123456789abcdef";
        push( &n);
        if ( digit = n/16)
                itoh( digit);
        putchar( hex[ n%16]);
        pop( &n);
}

push( unsigned int *p)
{
        fprintf( stderr,"push stivă %d:\tdigit=%d=%x\t
                &digit=%x\n",++nivrec, *p, *p, p);
}

pop( unsigned int *p)
{
        fprintf( stderr,"pop stivă %d:\tdigit=%d=%x\t
                &digit=%x\n",nivrec--, *p, *p, p);
}
```

Am utilizat două funcții pentru a vizualiza stiva: *push()* și *pop()*. Ambele afișează acțiunea și nivelul de recursivitate. Programul este scris de o asemenea manieră, încît permite salvarea conținutului stivei într-un fișier separat (fișierul salv), dacă se elimină marcajul de comentariu (altfel pe ecran). Execuția programului cu numărul *20347* ca argument în linia de comandă afișează:

```
push stivă 1:   digit=20347=4f7b    &digit=ffc8
push stivă 2:   digit=1271=4f7      &digit=ffbe
push stivă 3:   digit=79=4f         &digit=ffb4
```

push stivă 4: *digit=4=4* *&digit=ffaa*
pop stivă 4: *digit=4=4* *&digit=ffaa*
pop stivă 3: *digit=79=4f* *&digit=ffb4*
pop stivă 2: *digit=1271=4f7* *&digit=ffbe*
pop stivă 1: *digit=20347=4f7b* *&digit=ffc8*

Se observă că variabila *digit* are o adresă nouă la fiecare apel al funcției *itoh()* Cifrele hexazecimale se scot din stivă în ordine inversă, după cum arată și adresele din coloana a patra a afișării. Rezultatul conversiei este *4f7b*.

2.3.4. Transfer nelocal de control

Înainte de a încheia aspectele legate de zona stivă se analizează tehnica saltului în diferite zone ale unui program. Această tehnică schimbă, sub controlul programului, mediul de execuție.

Am prezentat în capitolul 1 posibilitatea saltului inter blocuri prin instrucțiunea goto. Turbo C limitează utilizarea acestei instrucțiuni în interiorul blocului unei funcții. Dar este posibil și transferul controlului unei alte funcții din program. Funcțiile *setjmp()* și *longjmp()* oferă această alternativă.

Execuția acestor funcții pare la început curioasă, deoarece nu se cunoaște modul cum ele utilizează mediul de execuție. În documentație găsim în fișierul **setjmp.h** declarațiile:

```
#include <setjmp.h>
...
int setjmp( env);
jmp_buf env;

și

void longjmp( env, val);
jmp_buf env;
int val;
```

Fișierul **setjmp.h** conține definiția tipului *jmp_buf*. Următorul program folosește concomitent funcțiile *setjmp()* și *longjmp()*. Inițial *setjmp()* este apelată pentru a salva mediul programului și returnează valoarea zero la primul apel. Altfel, *setjmp()* returnează valoarea celui de-al doilea argument al funcției *longjmp()*. Din această rațiune este necesară testarea valorii returnate de funcția *setjmp()*, pentru a detecta apariția unui transfer de control.

Problema salvării mediului programului de către funcția *setjmp()* este dependentă de partea hard a calculatorului, deoarece funcția accesează toți regiștrii acestuia și îi salvează în șirul *jmp_buf*.

Mecanismul de execuție al funcției *setjmp()* nu necesită salvarea cadrelor stivă.

Restaurarea mediului programului de către funcția *longjmp()* implică restaurarea regiștrilor salvați de către *setjmp()*, inclusiv a contorului programului. În felul acesta se forțează execuția unui alt set de instrucțiuni. În locul executării următoarei linii din program, se revine din apelul *setjmp()* cu o valoare diferită de zero. Schimbarea căii firești de execuție o numim **schimbare de context**. Un apel *longjmp()* nu mai permite revenirea în funcția apelantă.

Programul **jump.c** constituie un exemplu de utilizare al funcțiilor *setjmp()* și *longjmp()*. Programul citește de la tastatură un șir de caractere și îl memorează într-un buffer. Dimensiunea bufferului este de zece caractere. La întîlnirea caracterului linie nouă se face saltul la linia următoare. Dacă o linie conține mai mult de zece caractere sau se folosește o combinație specială de taste, conținutul curent al bufferului este ignorat. În aceste cazuri numărul liniei nu se modifică. Folosirea tastei *Esc* permite părăsirea programului. Programul conține trei funcții, codul funcției main() fiind următorul:

Programul 2.9.

```
/* jump.c - transfer nelocal de control */

#include <stdio.h>
#include <setjmp.h>

#define CTRLL '\014'
#define ESCK  '\033'
#define MAX   11

/* mediul ce se salvează sau se restaurează */
jmp_buf jmp_mediu;
void getlinie( char *);
int mygetch( void);

main()
{
        int retval;
        int linie = 1;
        char buf[ MAX], ch;
```

```
        retval = setjmp( jmp_mediu);
        if ( retval != 0) {
                switch ( retval) {
                case 1:
                        printf("\t linie prea lungă - se ignoră\n");
                        break;
                case 2: /* Ctrl-L */
                        printf("\t\t linia se ignoră\n");
                        break;
                case 3: /* Esc */
                        printf("\t\texit ? [d/]:");
                        ch = getch();
                        printf("%c\n", ch);
                        if ( ch=='d')
                                exit(1);
                }
        }
        printf("Linia %2d:\t", linie);
        while (1) {
                getlinie( buf);
                printf("Linia %2d:\t", ++linie);
        }
}
```

Inițial se apelează funcția *setjmp()*, care inițializează variabila *retval* cu valoarea zero. Ca atare, instrucțiunea *switch* nu se execută și se intră în bucla *while*, care prelucrează șirul de intrare. După afișarea numărului liniei curente se apelează funcția *getlinie()* pentru introducerea șirului de caractere. Codul acestei funcții este:

```
void getlinie( char *buf)
{
        int nc=0, ch;
        do {
                ch = mygetch();
                printf("%c", ch);
                if ( nc == MAX)
                        longjmp( jmp_mediu, 1);
                buf[ nc++]= ( ch == '\n') ? '\0' : ch;
        } while ( ch != '\n');
}
```


Șirul de intrare este preluat caracter cu caracter de către funcția *mygetch()*. Fiecare caracter citit este memorat în șirul *buf*. Excepție face caracterul linie nouă, în care caz se memorează în șir octetul *NULL* și se revine din funcție.

Dacă șirul de intrare conține mai mult de zece caractere se apelează funcția *longjump()*. La un astfel de apel execuția continuă cu funcția *setjmp()* din programul principal. Valoarea returnată de funcție este acum *1*, adică valoarea argumentului al doilea din funcția *longjmp()*. Programul afișează mesajul de ignorare a conținutului bufferului și se intră din nou în bucla *while*.

Funcția *mygetch()* conține două apeluri *longjmp()*. Combinația de taste *Ctrl-L* și tasta *Esc* transferă controlul în funcția principală la apelul funcției *setjmp()*. Valoarea returnată de *setjmp()* diferă în cele două cazuri și este tratată separat de instrucțiunea *switch*. Pentru *Ctrl-L* valoarea este *2*; se afișează un mesaj și se reintră în bucla de preluare a unei linii. Pentru *Esc* valoarea este *3* și dacă răspunsul la mesajul afișat este caracterul *'d'* programul se termină. În caz contrar programul continuă.

Codul funcției *mygetch()* este:

```
int mygetch( void)
{
        int c;
        c=getch();
        if ( c == CTRLL )
                longjmp( jmp_mediu, 2);
        if ( c == ESCK )
                longjmp( jmp_mediu, 3);
        return( c == '\r') ? '\n' : c;
}
```

La apelul funcției *longjmp()* sistemul a alocat pentru program cadrul stivă și variabilele *auto*. Un apel *setjmp()* salvează regiștrii interni cu aceste informații. După un apel *longjmp()* cadrul stivă este restaurat și variabilele *auto* își primesc vechile valori. Din acest motiv variabila *linie* nu-și schimbă valoarea după un apel *longjmp()*.

Execuția programului pe cîteva șiruri test afișează:

```
Linia 1: acceptată
Linia 2: corectă
Linia 3: prea lungă        linie prea lungă - se ignoră
Linia 3: rea<Ctrl-L>       linia se ignoră
```

Linia 3: *gata*
Linia 4: **<Esc>** exit ? [d/]: *n*
Linia 4: *ieşire*
Linia 5: **<Esc>** exit ? [d/]: *d*
>

Afişarea conţine trei coloane. Prima afişează linia curentă, a doua şirurile introduse, iar ultima mesajele informative. Programul acceptă prima şi a doua linie. Liniile a treia şi a patra sînt acceptate în final, iar în linia a cincea se încheie programul.

Transferul de control nelocal este foarte util în editoare de texte sau în aplicaţii cu meniuri şi ferestre, deoarece timpul de transfer al controlului este mic.

Mecanismul prezentat este puternic şi oferă o cale de evitare a mecanismelor obişnuite de control şi a celor cu fanioane, greoaie în utilizare şi predispuse la erori.

Se recomandă utilizarea cumpătată a funcţiilor *setjmp()* şi *longjmp()*. Prea multe apeluri ale acestora fac programele greu de urmărit.

2.4. Zona de date

Zona de date (numită zona datelor statice) conţine variabilele statice (interne sau externe) şi variabilele globale ale programului. Neexistînd nimic dinamic legat de aceste variabile, sistemul poate determina dimensiunea memoriei necesare la execuţia programului.

Zona de date este împărţită în două părţi: zona datelor iniţializate şi zona datelor neiniţializate. Zona de memorie în care se memorează o anumită variabilă depinde de declaraţia ei. Dacă declaraţia unei variabile statice sau globale este cu iniţializare, sistemul plasează variabila în zona datelor iniţializate; în caz contrar plasînd-o în zona datelor neiniţializate. Zona datelor neiniţializate este numită *BSS* (" Block Started by Symbol") şi este iniţializată de sistem cu valoarea zero.

Deoarece sistemul determină dimensiunea şi valoarea acestor variabile, aceste variabile poartă numele de variabile de încărcare. Ele există din momentul încărcării programului în memorie şi pînă la sfîrşitul lui. La alocarea memoriei pentru variabilele de încărcare, sistemul actualizează variabilele iniţializate cu valorile lor şi iniţializează zona *BSS* cu zero.

Cunoaşterea modului în care un program foloseşte zona de date este de folos în determinarea necesarului de date.

2.4.1. Alocarea

Compilatorul memorează în zona de date:
- variabilele statice interne şi externe,
- variabilele globale,
- şiruri şi structuri iniţializate,
- constante şir.

Variabilele globale sînt declarate în afara funcţiilor şi pot fi utilizate şi de alte fişiere. Şirurile şi structurile iniţializate din blocuri sau funcţii se găsesc în zona de date, dacă declaraţia lor este precedată de cuvîntul cheie *static*.

Constantele şir, funcţie de compilator, pot fi plasate în zona de date sau în zona text. Pentru început presupunem că ele se vor găsi în zona de date.

Exemplul de mai jos ilustrează modul în care este folosită zona de date:

Programul 2.10.

```
/* data.c - alocarea memoriei în zona de date */

static int status;  /* static extern */
char buf[10];       /* şir extern - global */
strct complex {   /* şir extern - structură globală */
        float real, imag;
} polar;

main()
{
/* static intern - long */
        static long f;
/* static intern - şir */
        static float vals[] = { 1.2, 3.4, 5.6};
/* static intern - structură */
        static struct complex plan;
/* constantă şir */
        printf("%f %f\n", vals[1], vals[2]);
}
```

Variabilele *stare*, *buf* şi *polar* se află în zona de date, deoarece ele sînt declarate în afara funcţiei *main()*. Variabilele static interne *flag*, *val* şi *plan* şi constanta şir "*%f %f\n*" se vor găsi tot în zona de date.

2.4.2. Zona BSS

Variabilele statice neinițializate formează zona *BSS*. Din acest punct de vedere exemplul anterior arată astfel:

```
static int status;  /* BSS extern static - întreg global */
char buf[10];       /* BSS extern - şir întreg global */
struct complex {
        float real, imag;
} polar;            /* BSS extern - structură globală */

main()
{
/* BSS static intern - long */
        static long f;
/* static intern - şir */
        static float vals[] = { 1.2, 3.4, 5.6};
/* BSS static intern - structură */
        static struct complex plan;
/* constantă şir */
        printf("%f %f\n", vals[1], vals[2]);
}
```

Exceptînd şirul *val*, toate variabilele se află în zona *BSS*. Constanta şir nu e plasată niciodată în zona *BSS*, deoarece ea trebuie inițializată. Domeniul de existență al unei variabile n-are legătură cu zona *BSS*.

Se constată că separarea variabilelor inițializate de cele neinițializate este necesară mai mult pentru încărcător decît pentru compilator. Se cunoaşte că înaintea execuției C-ul inițializează la zero variabilele statice neinițializate şi variabilele globale ale programului. Pentru a realiza acest lucru, compilatorul marchează aceste variabile astfel ca încărcătorul să le poată grupa în momentul linkeditării. La momentul execuției, sistemul încarcă variabilele *BSS* într-o zonă a memoriei şi umple toată zona cu valoarea zero. Această modalitate este oricum mai rapidă decît inițializarea pe rînd a fiecărei variabile.

Avantajul utilizării variabilelor de clasă de memorie *static* se poate constata pe cîteva exemple. Fie codul următor:

```
#define MAX 1000
int buffer[MAX];        /* BSS */
```



```
main()
{
        int i;
        /* inițializarea */
        for ( i=0; i<MAX; i++)
                buffer[i]=0;
        ...
}
```

se utilizează bucla *for* pentru inițializarea elementelor şirului cu valoarea zero. Inițializarea, pe lîngă faptul că este lentă, este şi redundantă. Codul:

```
int număr (int *buf, int nelem, int val)
{
        int i;
        int nc=0;           /* auto */
        for ( i=0; i<nelem; i++)
                if (buf[i]==val)
                        nc++;
        return nc;
}
```

memorează în variabila *nc*, de clasă de memorie *auto*, numărul valorilor unui şir. Înainte de a parcurge şirul, variabila *nc* este inițializată cu valoarea *0*. La revenirea din funcție variabila *nc* dispare de pe stivă, iar următorul apel al funcției inițializează din nou variabila cu *0*. Ca atare, nu se poate utiliza valoarea returnată de funcție ca o valoare cu caracter global pentru program, deoarece ea se reinițializează după fiecare apel.

Menținerea structurii funcției impune definirea unei noi variabile, *total*, pentru a contoriza corect valorile.

```
#define MAX 1000
int date[MAX]
int total=0;        /* auto */
...
        while (1) {
                ...
        total += număr(date, MAX, 0x7f);
        printf ("%d\n", total);
        ...
        }
```

O soluție mai eficientă constă în mutarea variabilei *total* în corpul funcției *număr()* și schimbarea clasei ei de memorie.

```
int număr(int *buf, int nelem, int val)
{
        int i;
        static int total;
        for ( i=0; i<nelem; i++)
                if (buf[i]==val)
                        total++;
        return total;
}
```

Întrucît variabila *total* este acum în *BSS*, valoarea ei este zero la primul apel și este mereu actualizată funcție de valoarea ei anterioară. Modificarea ce trebuie operată în programul principal este:

```
#define MAX 1000
int date [MAX];
...
while (1){
        ...
        printf("%d\n", număr(date, MAX, 0x7f));
        ...
}
```

2.4.3. Inițializare

Zona datelor inițializate conține variabilele statice și globale, care se inițializează explicit în programul C. Înainte de execuția programului sistemul inițializează fiecare variabilă în memorie. Constantele șir, dacă sînt memorate în zona de date, apar în zona datelor inițializate.

Programul 2.11.

```
/* initd.c -alocă memoria pentru date inițializate */

#include <stdio.h>
static int status=-1;       /* static extern - întreg */
char buf[]="dar";           /* extern - șir global */
```



```
struct complex {            /* extern - structură globală */
        float real, imag;
} polar = { -1.7, 2.4 };

main()
{
/* static intern - long */
        static long f;
/* static intern - șir */
        static float vals[] = { 1.2, 3.4, 5.6};
/* static intern - structură */
        static struct complex plan;
/* constantă șir */
        printf("%f %f\n", vals[1], vals[2]);
}
```

Toate variabilele sînt inițializate; șirul constant din funcția *printf()* apărînd în zona datelor neinițializate. Diferența dintre clasele de memorie *static* și *auto* o punem în evidență prin segmentul de program:

```
{
        int var = 3;            /* auto */
        static int variab = 4;  /* static intern */
        ...
}
```

Întrucît variabila *var* are clasa de memorie *auto*, compilatorul alocă spațiu pe stivă și generează cod pentru încărcarea valorii *3* în locația corespunzătoare variabilei. Dar *variab*, avînd clasa de memorie *static*, se găsește în zona de date inițializate, deci inițializarea ei este sarcina sistemului și nu cea a compilatorului. Valoarea *4* este încărcată în *variab* înainte de execuția programului. Rezultă, așadar, că inițializarea variabilelor statice este mai rapidă, deoarece compilatorul nu generează cod pentru inițializare.

Declarația:

```
static char șir[] = "OK";
```

nu generează cod pentru încărcarea șirului constant "*OK*" în variabila *șir*, iar caracterele șirului se află în zona de date înainte de execuția programului:

Fie programul:

Programul 2.12.

```
/* loop.c - var auto */

main()
{
        int i;
        for ( i=0; i<3; i++) {
                int j=10;
                printf("i= %d \t j= %d\n", i, j);
                j++;
        }
}
```

Execuţia afişează:

i= 0 j= 10
i= 1 j= 10
i= 2 j= 10

Valoarea variabilei *j* nu se modifică, deoarece iniţializarea ei se face la fiecare intrare în buclă. Compilatorul alocă pentru ea spaţiu stivă o singură dată. Dacă variabila *j* ar avea clasa de memorie *static*, la execuţie nu s-ar genera cod pentru iniţializarea ei. Sistemul iniţializează variabila *j* cu *10* la prima intrare în buclă, după care valoarea ei anterioară este incrementată cu 1 de două ori.

Programul 2.13.

```
/* loop2.c - variabile statice */

main()
{
        int i;
        for ( i=0; i<3; i++) {
                static int j=10;
                printf("i= %d \t j= %d\n", i, j);
                j++;
        }
}
```

>loop2

Execuţia afişează:

i= 0 j= 10
i= 1 j= 11
i= 2 j= 12

2.4.4. Constante şir

Turbo C permite utilizarea constantelor caracter şi a constantelor şir. Dacă un caracter Dacă un caracter ASCII este inclus constantă şi nu o locaţie de memorie. De exemplu, compilatorul generează numărul hexazecimal *6b* pentru caracterul *'k'*.

Constantele şir au dimensiunea variabilă, compilatorul alocînd cîte un octet pentru fiecare caracter plus octetul de sfîrşit (octetul *NULL*).

C permite utilizarea constantelor şir la:
- iniţializarea pointerilor la caracter,
- atribuirea pointerilor la caracter,
- transmiterea argumentelor în funcţii,
- referinţe prin pointer caracter.

Primele trei utilizări sînt ilustrate de exemplele:

```
char * prenume = "Adrian";        /* 1 */
*prenume = "Gabriel";             /* 2 */
puts("Daniela");          /* argument de funcţie */
```

Constante şir apar şi la iniţializarea şirurilor de caractere, ca în:

```
char copil[] = "Radu";
```

dar atribuirea este notaţia prescurtată pentru:

```
char copil[] = { 'R','a','d','u','\0'};
```

şi nu e tratată de compilator ca o atribuire de constantă şir.

Ultima utilizare necesită mai multe explicaţii. Fie exemplul:

Programul 2.14.

```
/* strconst.c - utilizarea constantei şir */

#include <stdio.h>

main()
{
        printf("%s\n", "turbo C v-2.0"+6);
        printf("%c\n", *("12345"+3));
        printf("%c\n", "12345"[1]);
}
```

Execuţia afişează:

```
C v-2.0
4
2
```

Utilizarea constantelor şir în locul pointerilor la caracter nu e uzuală. Expresia *"turbo C v-2.0" + 6* este evaluată la adresa caracterului al şaselea al constantei şir. Expresia **("12345" +3)* se evaluează la *'4'* prin aplicarea operatorului de indirectare pe adresa celui de-al patrulea element al constantei şir *"12345"*.
Ultima expresie *"12345"[1]* se evaluează la *'2'* prin indexarea constantei şir. Constanta şir *"1234"* e utilizată în acest caz în locul numelui şirului.

Rezultă că utilizînd o constantă şir se poate îmbunătăţi codul funcţiei recursive *itoh()* anterior prezentate. Se constată că această funcţie alocă pe stivă, la fiecare apel al ei, spaţiu pentru variabila *hex*. Prin utilizarea constantei şir în locul pointerului *hex*, codul funcţiei devine:

```
itoh( unsigned int n)
{
        int digit;

        push( &n);
        if (digit = n/16)
                itoh( digit);
        putchar( "0123456789abcdef"[ n%16]);
        pop( &n);
}
```


Se elimină astfel pointerul din apelul recursiv şi se cîştigă spaţiu stivă. O altă soluţie o constituie utilizarea declaraţiei:

```
static char *hex = "0123456789abcdef";
```

ce păstrează structura programului iniţial şi spaţiul stivă cîştigat.

Fie un program ce conţine instrucţiunile:

```
char buf[] = "şir";
char *p = " ";
```

Ele se pot găsi oriunde în program, dar în baza celor discutate se impun precizările: instrucţiunea ce iniţializează şirul *buf* este exterioară oricărei funcţii, *buf* e adresa unui şir de caractere de lungime *4* şi numele şirului este o constantă. Ultimul aspect e foarte important, deoarece o atribuire nouă, de genul:

```
char buf[] = " şir nou "
```

va produce o eroare la compilare. (Este ca şi cum am încerca atribuirea *5=6.*) Pe de altă parte, pointerul *p* poate avea clasa de memorie *auto* sau *static*. În acest caz *p* este iniţializat cu o constantă şir.
Dacă se modifică valoarea lui *p*, se modifică adresa la constanta şir. Deci, o atribuire de genul:

```
p= "şir nou"; este perfect legală.
```

Presupunem programul:

Programul 2.15.

```
/* şir.c - ilustrează constante şir */

char buf[] = "mesaj";
char *p = "şir_vechi";

main()
{
        char *q = "şir";
        printf("adresa buf: %x\n", buf);
        printf("adresa q : %x\n", &q);
        printf("adresa p : %x\n", &p);
```

```
    printf("conținut \t\t lungime \t\t adresă\n");
    printf("%s\t\t\t %d \t\t\t %x\n", buf, strlen( buf), buf);
    printf("%s\t\t %d \t\t\t %x\n", p, strlen( p), p);
    printf("%s\t\t\t %d \t\t\t %x\n", q, strlen( q), q);
    p = "șir_nou";
    buf[ 2] = 'n';
    printf("%s\t\t\t %d \t\t\t %x\n", buf, strlen( buf), buf);
    printf("%s\t\t\t %d \t\t\t %x\n", p, strlen( p), p);
}
```

Execuția programului generează ieșirea:

adresa buf: 194
adresa p : ffd4
adresa q : 19a

conținut	*lungime*	*adresa*
mesaj	*5*	*194*
șir_vechi	*9*	*19c*
șir	*3*	*1a6*
menaj	*5*	*194*
șir_nou	*7*	*22d*

Înainte de execuție zona de date a programului arată astfel:

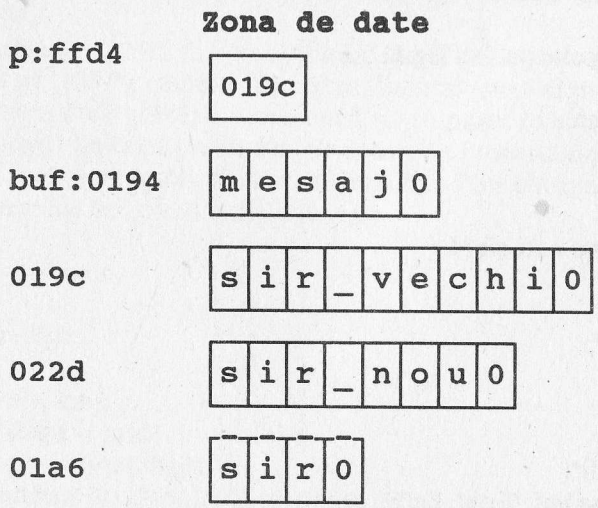


Figura 2.4. Starea șirurilor la încărcare

Sistemul alocă spațiu pentru șirurile "*șir*" și "*șir_nou*" în zona de date, dar nimic nu le referă. După alocarea lui *q* pe stivă, figura devine:

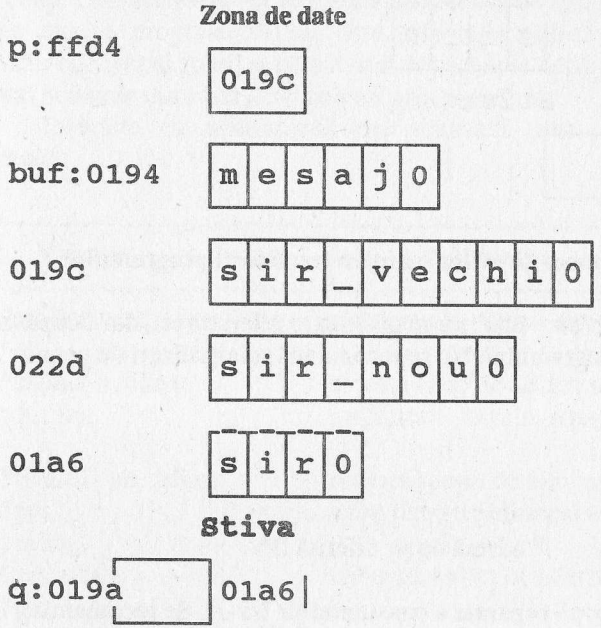
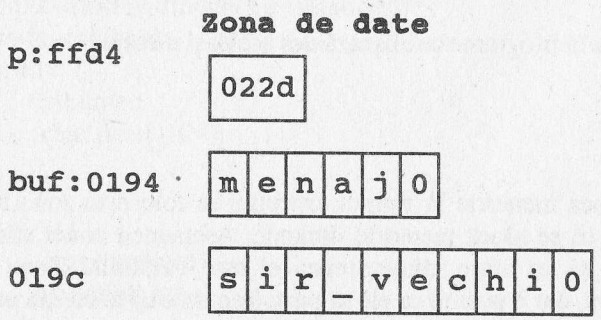


Figura 2.5. Starea șirurilor la execuție

Pointerului *q* i se atribuie adresa șirului "*șir*" (*01a6*). Compilatorul atribuie adresa șirului din zona de date conținutului locației de pe stivă, realizînd atribuirea pointerului. Figura următoare ilustrează alocarea înainte de terminarea programului. Acum *p* referă șirul "*șir_nou*" iar *buf* e modificat.



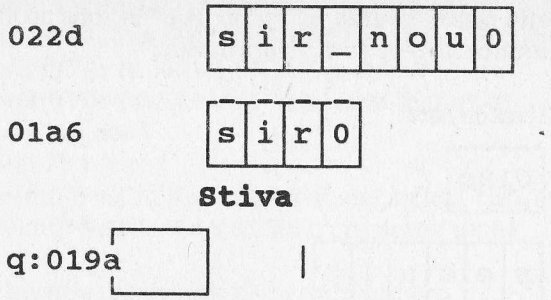


Figura 2.6. Starea şirurilor înaintea terminării programului

Constanta şir "*şir_vechi*" nu se mai poate referi direct, dar ocupă memorie pînă la terminarea programului. Nu se recomandă reiniţializări de genul:

```
char *p = "şir-1";
...
p = "şir-2";
...
p = "şir-1";        /* adresă de şir diferită !!*/
```

deoarece se creează copii separate a constantei şir "*şir-1*". Se recomandă:

```
char *p ="şir-1";
char *psalv;
...
psalv=p;            /* salvează adresa şir-1 */
p="şir-2";
...
p=psalv;            /* restaurarea adresei şir-1 */
```

Tehnica este utilizată în programe ce afişează des aceleaşi mesaje.

2.5. Zona Heap

Pentru a aloca memorie în timpul execuţiei se foloseşte zona heap. Se spune, în acest caz, că se alocă memorie dinamic. Asemenea zonei stivă, zona heap este dinamică şi, ca atare, dimensiunea ei este variabilă. Zona heap e separată de zona stivă, dar e posibil ca ele să partajeze aceeaşi zonă din memorie.

În această situaţie, avînd în vedere că stiva creşte spre adrese de memorie mici, iar zona heap creşte spre adrese de memorie mari, este foarte posibil ca un program ce foloseşte memorie multă pentru stivă, să nu mai dispună de memorie pentru zona heap. Este adevărată şi reciproca acestei observaţii. Gestionarea zonei heap rămîne sarcina programatorului, care prin intermediul funcţiilor de bibliotecă alocă sau eliberează spaţii şi detectează eventualele depăşiri ale capacităţii zonei. Accesul la datele din zona heap se face prin pointeri.

Interfaţa cu utilizatorul este realizată prin următoarele funcţii de bibliotecă:

Funcţie	Acţiune
void *malloc(size_t dim)	alocă spaţiu pentru *dim* octeţi
void *calloc(n, dim) size_t n, dim;	alocă *n* articole de *dim* octeţi iniţializate cu valoarea zero
void *realloc(ph, ndim) void *ph; size_t ndim;	realocă spaţiu de dimensiune *ndim* pentru pointerul vechi *ph*
void free(ph) void *ph;	eliberează spaţiul referit de *ph*

Tabelul 2.11.

Următorul program alocă memorie pentru structură *bloc*.

Programul 2.16.

```
/* bloc1.c- alocare dinamică a memoriei */
#include <stdio.h>
struct bloc {
        int antet;
        char date[1024];
};
main()
{
        char *malloc();
        struct bloc *p;
```

```
        p=(struct bloc *) malloc( sizeof( struct bloc));

        if ( p == ( struct bloc *) NULL) {
                fprintf( stderr, "nu se poate aloca spațiu\n");
                exit(1);
        }
}
```

Operatorul *sizeof* determină numărul de octeți necesari structurii, iar funcția *malloc()* returnează adresa unde s-a făcut alocarea. Este necesar castul pentru a echivala tipul pointer la void cu tipul pointer la structura *bloc*. Testul valorii returnate de funcția *malloc()* certifică dacă s-a putut face alocarea.

Funcția *calloc()* e similară funcției *malloc()*, dar zona alocată e inițializată cu valoarea zero. Acesta este motivul pentru care funcția *calloc()* se execută mai încet decît funcția *malloc()*.

Funcția *realloc()* permite schimbarea dimensiunii unei zone alocate anterior. Primul argument din apel este pointerul la zona alocată, iar al doilea este noua dimensiune a zonei (în octeți). Dacă nu există spațiu contiguu la realocare, funcția *realloc()* returnează o adresă diferită de cea anterioară. Aceasta implică mutarea de către funcție a datelor la o adresă nouă. Rezultă că timpul de execuție al unui apel *realloc()* crește dacă este necesară mutarea datelor. Ocazional *realloc()* poate să returneze aceeași adresă. Acest lucru e important cînd se utilizează tabele de evidență a spațiului alocat, deoarece în acest caz realocările impun mereu actualizarea tabelelor.

Următorul program demonstrează utilitatea funcției *realloc()* în cazurile în care nu se cunoaște dinainte dimensiunea spațiului ce trebuie alocat.

Programul citește de la consolă un șir de caractere și alocă memorie heap de atîția octeți cît este dimensiunea șirului.

Programul 2.17.

```
/* citin.c - citește un șir variabil din intrare */
#include <stdio.h>
#define MAX 256

main()
{
        char *p, *citin();
        p = citin(">>");
        printf("sirul din intrare: %s\tlungime = %d\n", p, strlen(p));
}
```



```
char *citin( char *prompt)
{
        char *p, *q;
        int nc;
        unsigned size = MAX;
        char *malloc(), *realloc();
        if (( p = q = malloc( size)) == NULL) {
                printf("eroare malloc()\n");
                exit(1);
        }
        printf("%s", prompt);
        for ( nc=1; ( *p = getchar()) != '\n' && *p != EOF; nc++) {
                if ( nc % MAX == 0) {
                        size += MAX;
                        if (( p=q= realloc( q, size)) == NULL) {
                                printf("eroare relloc()\n");
                                exit(1);
                        }
                        p += nc;
                        continue;
                }
                p++;
        }
        *p = '\0';
        return realloc(q, nc);
}
```

Funcția *citin()* alocă inițial *256* de octeți în memoria heap. Variabila *nc* contorizează numărul de caractere citite și memorate în zona heap prin **p*. Cînd s-au citit *256* de caractere se realocă memorie heap pentru alte *256* de caractere, transmițînd pointerul original *q* înapoi funcției *realloc()*. După fiecare relocare se incrementează pointerul *p* cu valoarea variabilei *nc*, pentru ca pointerul să refere primul caracter neutilizat al zonei nou alocate.

Înainte de a reveni din funcție, se completează șirul de intrare cu octetul *NULL* și se apelează funcția *realloc()* cu numărul de caractere citite, pentru eliberarea spațiului neutilizat.

Se constată că programul afișează șirul introdus și lungimea sa.

În program am actualizat constanta *MAX* cu *256*. Valoarea e mare în raport cu numărul de caractere citite, dar ea reduce apelul prea des al funcției *realloc()*. De fapt, cu cît mai mare este constanta *MAX*, cu atît mai rar este apelul funcției *realloc()*.

Eliberarea spațiului alocat în memoria heap se realizează prin apelul funcției *free()*. Analiza acestei funcții se va face în capitolele 5 și 6.

2.5.1. Inițializarea variabilelor din program

Programul următor semnalează eroare la compilare:

```
main()
{
        char buf[5];            /* stivă */
        static char *p=buf;     /* dată inițializată */
}
```

Greșeala e datorată claselor de memorie atașate variabilelor. Șirul *buf* are clasă de memorie *auto* și este creat pe stivă la execuția programului, iar variabila *p* este un pointer în zona de date. Din moment ce *p* este în zona de date, valoarea sa inițială trebuie determinată înainte de execuția programului. Compilatorul nu poate determina valoarea sa inițială, deoarece *buf* încă nu există; se semnalează eroare de utilizare a clasei de memorie.

Exemplul ce inițializează corect pointerul este:

```
main()
{
        static char buf[5];     /* BSS */
        static char *p=buf;     /* dată inițializată */
}
```

Mediul de execuție:

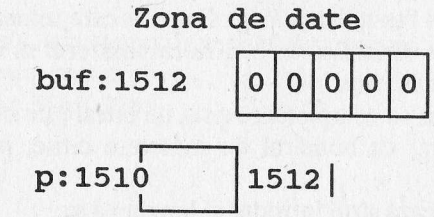


Figura 2.8. Inițializarea variabilelor statice

Atît *p* cît și *buf* există în zona de date. Șirul *buf* se află în zona *BSS*, întrucît nu este inițializat. Adresa lui *buf* (*1512*) este o constantă care e cunoscută în momentul încărcării, iar pointerul *p* referă primul element al șirului. Compilatorul nu generează cod pentru inițializare.

Următorul exemplu ilustrează inițializarea unui pointer de clasă de memorie *auto* la o variabilă de aceeași clasă de memorie.

```
main()
{
        char buf[5];    /* Stivă */
        char *p=buf;    /* Stivă */
}
```

În acest caz, mediul de execuție este:

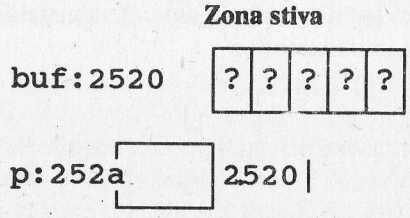


Figura 2.9. Inițializarea variabilelor auto

Atît *buf* cît și *p* sînt variabile *auto* pe stivă. Pentru a inițializa pointerul *p*, *buf* trebuie să existe. Deci inițializarea trebuie să urmeze declarației șirului *buf*. Acest lucru permite compilatorului generarea codului pentru încărcarea adresei șirului *buf* în *p*, întrucît adresa șirului *buf* din cadrul stivă e cunoscută cînd *p* este inițializat. După cum se observă din figură, elementele șirului *buf* sînt neinițializate (au valorile găsite în cadrul stivă la momentul execuției).

Pentru inițializarea variabilelor de clasă de memorie diferită, se consideră programul:

```
main()
{
        static char buf[5];     /* BSS */
        char *p=buf;            /* Stivă */
}
```

Mediul de execuție:

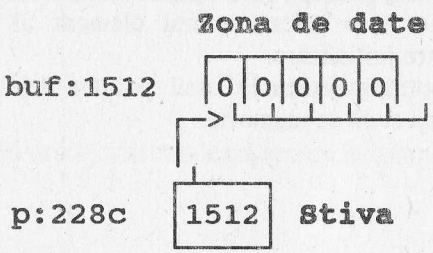


Figura 2.10. Inițializarea variabilelor statice și auto

Adresa lui *buf* este în *BSS* și fiecare element are valoare zero. În momentul execuției compilatorul alocă spațiu stivă pentru *p* și generează codul pentru inițializarea lui cu adresa șirului *buf* din zona de date. Se constată deci că inițiaiizarea pointerului se face în momentul execuției.

Următorul exemplu:

```
int bloc=1024;  /* dată inițializată */
main()
{
        static int ori=10 * bloc;  /* dată inițializată */
}
```

generează eroare la compilare, deoarece programul încearcă să inițializeze variabila *ori* cu o expresie, folosind valoarea variabilei *bloc*. La inițializarea unei variabile statice trebuie utilizate constante sau adrese.

Schimbînd clasa de memorie la *auto*, compilarea decurge fără erori.

Expresiile cu variabile pot fi folosite, însă, la inițializarea variabilelor de clasă de memorie auto. Atunci se poate muta declarația variabilei *bloc* în funcția *main()* și compilarea este fără erori.

Dacă nu dorim schimbarea clasei de memorie a variabilei *ori*, putem proceda astfel:

```
#define BDIM 1024
int bloc=BDIM;
main()
{
        static int ori=10 * BDIM;          /* stivă */
}
```


Compilatorul acceptă aceasta, deoarece variabila statică *ori* este inițializată cu o expresie constantă.

2.5.2. Inițializarea variabilelor prin funcții

Să presupunem declarația unei variabile și inițializarea ei cu valoarea returnată de o funcție. Dacă variabila este de clasă *auto*, inițializarea e posibilă după cum urmează:

```
main( int argc, char *argv[] )
{
        int lung = strlen( argv[1]);
}
```

Variabila *lung* e inițializată cu lungimea șirului ce apare ca prim argument în linia de comandă. Dacă ea ar fi declarată *static*, inițializarea:

```
        static int lung = strlen( argv[1]);
```

ar produce eroare la compilare, deoarece nu se poate inițializa o variabilă statică cu valoarea altei variabile statice. Ambele acțiuni se produc la execuție. Dacă programul necesită totuși ca variabila *lung* să fie statică, există posibilitatea:

```
main( int argc, char *argv[] )
{
        static int lung;                /* BSS */
        lung = strlen( argv[1]);
}
```

În acest caz, valoarea inițială a variabilei *lung* este zero. În timpul execuției instrucțiunii de atribuire, compilatorul generează cod pentru încărcarea variabilei *lung* cu valoarea returnată de funcția *strlen()*.

Pointerii la funcții de clasă *auto* sau *static* pot fi inițializați, deoarece adresele funcțiilor sînt cunoscute în momentul încărcării (nu e necesară o atribuire separată).

```
main()
{
        int f();
        int (*p)()=f;                   /* auto */
        static int (*q)();              /* date inițializate */
}
```

```
f()
{
        ...
}
```

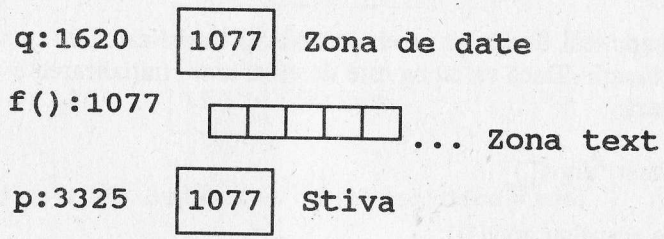


Figura 2.11. **Inițializarea variabilelor statice și auto cu adrese din zona text.**

Să analizăm modul în care clasa de memorie și mediul de execuție pot influența valoarea returnată din funcții. Pentru aceasta, să considerăm funcția *itoh()* într-o versiune nerecursivă:

Programul 2.18.

```
/* itoh4.c - conversie hexa nerecursivă eroare la execuție */

#include <stdio.h>

main()
{
        unsigned int num=1199;
        char *itoh();
        printf("%5d = %s h\n", num, itoh( num));
        printf("%5d = %s h\n", 32767, itoh(32767));
}

char *itoh( unsigned int num)
{
        int digit, nd=4;
        char buf[5];
        buf[nd]='\0';
```



```
        while (nd) {
                digit=num%16;
                buf[--nd]=(digit <= 9) ? digit+'0' : digit+'a'-10;
                num /=16;
        }
        return buf;
}
```

Șirul *buf* conține patru caractere plus octetul *NULL*. Funcția *itoh()* nefiind recursivă, pentru a realiza conversia corectă, cifrele trebuie memorate în ordine inversă în șirul rezultat. Totul pare corect, dar rezultatul execuției demonstrează contrariul.

Plasarea unor instrucțiuni de depanare în codul sursă nu indică vreo eroare de algoritm. Eroarea este în linia:

```
        return buf;
```

ce returnează adresa șirului.
Șirul *buf* este de clasă de memorie *auto* și se află pe stivă. La revenirea din funcție, însă, cadrul stivă este înlocuit cu cel anterior și, ca atare, șirul dispare.
Observația ce se impune în acest caz este de a nu se returna adrese de variabile *auto* din funcții.

Returnînd o valoare și nu o adresă, programul este corect întotdeauna, deoarece compilatorul copiază variabila *auto* din cadrul stivă într-un registru sau într-o locație de memorie. Ca atare, dispariția cadrului stivă nu va afecta valoarea returnată.

Corectarea erorii se poate face prin alocare dinamică, cu returnarea unui pointer la cifrele conversiei.

O soluție mai eficientă rămîne însă declararea statică a șirului *buf*. Codul funcției devine:

Programul 2.19.

```
/* itoh5.c - conversie hexa nerecursivă */

#include <stdio.h>

main()
{
        unsigned int num=1199;
        char *itoh();
```

```
        printf("%5d = %s h\n", num, itoh(num));
        printf("%5d = %s h\n", 32767, itoh(32767));
}

char *itoh( unsigned int num)
{
        int digit, nd=4;
        static char buf[5];

        while (nd) {
                digit=num%16;
                buf[--nd] = (digit <= 9) ? digit+'0' : digit+'a'-10;
                num /=16;
        }
        return buf;
}
```

Încheiem acest capitol cu un program ce conține toate tipurile de variabile întîlnite.

Programul 2.20.

```
/* map.c */

#include <stdio.h>

int P=0x36c;    /* dată globală inițializată */
char c;         /* global BSS */
static char neg=-1;/* dată locală inițializată*/

main()
{
        int i=0;                /* stiva */
        char *malloc();

        static int m;           /* BSS */
        static int k=20;        /* data initializata */
        static int *ip=&k;      /* data initializata */
        static char b[10];      /* BSS */
        static char *q=b;       /* data initializata */
```



```
        char d[10];             /* stiva */
        char *p=d;              /* stiva */
        char *pheap=            /* stiva */
                malloc( 100);   /* heap */

        static char tx[]="Prog"; /* data inițializată */

        struct articol {
                int dim;
                char *str;
        } art[10];              /* stiva */

        struct articol *ps=(struct articol *)
                malloc(sizeof(struct articol)); /* heap */

        if ( k>10) {            /* stiva */
                int j=1;
        }
}
```

Date initializate		BSS	
Variabile	Valori initiale	Variabile	Valori initiale
P	36c	c	0
n	-1	m	0
k	20	b	0
ip	&k		
q	&b		
tx	Prog		

Zona stiva		Zona heap	
Variabile	Valori initiale	Adresa	Numar octeti
i	0	ph	100
d	?	ps	40
p	&d		
ar	?		
j	1		
ph	7e00		
ps	8012		

2.6. Probleme propuse pentru rezolvare

1. Să se scrie un program (**calc.c**), care citește din linia de comandă numele unei funcții și numărul de parametri asociați ei. Se va utiliza structura:

```
struct matem {
        char *nume;
        double (*pf)();
        int narg;
};
```

pentru crearea unui tabel cu funcții matematice din biblioteca TC.

Se vor folosi doar acele funcții ce necesită argumente de tip *double* și returnează valori de tip *double*. Execuția programului pe cîteva date test arată astfel:

```
>calc sqrt 25
5
>calc pow 2.5 3.5
24.7053
```



```
>calc exp .05
1.64872
```

2. Să se scrie o funcție `itoc()` pentru conversia unui întreg fără semn la un șir de caractere binare, octale și hexazecimale. Baza de conversie se transmite ca parametru funcției *itoc()*.
3. Să se rescrie *itoc()* ca funcție recursivă.
4. Programul **itoh3.c** utiliza funcția *putchar()* pentru afișarea unui caracter. Considerînd că memorarea caracterelor se face recursiv într-un șir, se cere să se identifice eroarea din programul:

```
/* itoh6.c - conversie hexa recursiva */

#include <stdio.h>

char * itoh( unsigned int n, char *s);

main( int argc, char *argv[])
{
        unsigned int num;
        char buf[10];
        if ( argc == 1) {
                fprintf(stderr, "utilizare: %s numar\n", argv[0]);
                exit(1);
        }
        num=atoi( argv[1]);

        itoh( num, buf);
        printf("%s h\n", buf);
}
char * itoh( unsigned int n, char *s)
{
        int digit, i;
        static char *hex="0123456789abcdef";

        if ( digit=n/16)
                itoh( digit, s);
        s[i++]=hex[ n % 16];
}
```

Explicați ce se întîmplă la execuție. Să se corecteze programul.

5. Să se modifice programul **jump.c**, astfel încît apăsarea tastei **Del** să permită terminarea execuției programului.

6. Utilizînd tehnica prezentată în programul **citin.c**, să se scrie două funcții identice ca acțiune cu funcțiile de bibliotecă *gets()* și *fgets()*. Formatul lor este:

```
char *getșir();
char *fgetșir( int n, FILE *infile);
```

Versiunile noi de funcții nu vor necesita un tampon pentru memorarea datelor de intrare. Ambele funcții returnează un pointer în zona heap unde se află alocate caracterele efectiv citite.

7. Să se analizeze programul:

```
/* concat.c - concatenare șiruri */

#include <stdio.h>

main()
{
        char *join();
        printf("%s\n", join("port","avion"));
}

char *join( char *s1, char *s2)
{
        char *strcat(), *strcpy();
        struct {
                char buf[256];
        } sir;
        return strcat(strcpy( sir.buf, s1), s2);
}
```

Programul concatenează două șiruri. Compilați și executați programul. Să se explice eventualele anomalii.

3. Alternative și posibilități

3.1. Șiruri și pointeri

Înainte de a aborda expresii complicate cu pointeri și șiruri multidimensionale avem nevoie de o bază temeinică asupra lucrului cu aceste noțiuni. Conceptele fundamentale pe care dorim să le punem în evidență ne oferă instrumentele necesare scrierii unui cod C eficient și înțelegerii viitoarelor programe.

3.1.1. Reguli de bază

Fie declarațiile:

```
char a[] = "exemplificare";      /* șir de char */
char *p = a;                     /* pointer la șir */
```

Figura 3.1. ilustrează modul în care s-a făcut alocarea memoriei pentru șirul *a* și pointerul *p* (s-a presupus că ambele valori sînt în zona de date și adresa este pe *16* biți).

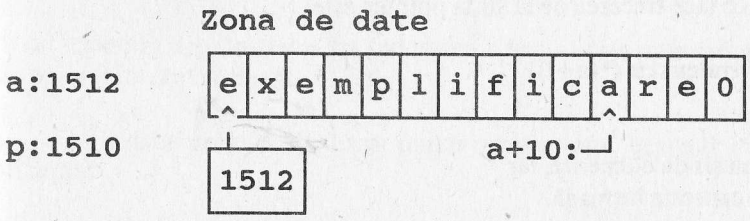


Fig. 3.1. Șir și pointer

Compilatorul tratează numele șirurilor ca și constante. Aceasta presupune că variabila șir *a* poate fi utilizată oriunde un pointer este permis, exceptînd partea stîngă a unei expresii de atribuire și combinarea lui cu operatorii de incrementare și decrementare (ex. *a++*). Următoarele instrucțiuni:

```
printf("%s\n", a);       /* numele șirului */
printf("%s\n", p);       /* pointerul la șir */
```

afișează pe două linii separate șirul "*exemplificare*". Primul apel convertește numele șirului la un pointer și compilatorul transmite adresa *1512* funcției *printf()* (figura 3.1). În al doilea apel, *p* fiind pointer, compilatorul transmite funcției *printf()* conținutul locației variabilei *p* (*1510*), adică tot adresa *1512*.

Instrucțiunile:

```
printf("%s\n", a+10);
printf("%s\n", p+10);
```

afișează pe două linii separate șirul "*are*". Cele două expresii le numim *pointeri cu deplasament*. În prima instrucțiune, *a+10* referă al unsprezecelea caracter din șir. Compilatorul calculează adresa *1522* prin adunarea constantei *10* la adresa șirului *1512*. Instrucțiunea a doua este similară, dar utilizează pointerul *p*. Pointerii cu deplasament pot utiliza atît numele șirului cît și pointeri la șir.

Similar, instrucțiunile:

```
putchar( a[10]);
putchar( p[10]);
```

afișează caracterul '*a*' de două ori. Instrucțiunea a doua ilustrează posibilitatea indexării unui pointer la șir.

Aceste exemple pun în evidență existența unei strînse legături între șiruri și pointeri. Întotdeauna compilatorul tratează numele șirurilor ca pointeri. Regula de bază ce face trecerea de la șir la pointer este:

a[i] echivalent cu (*(a+i))

unde:

a este un șir de elemente, iar
i este o expresie întreagă.

Regula de bază este **independentă** de tipul șirului *a*. Perechea de paranteze exterioare este redundantă în acest caz, dar nu și dacă *a* este un șir multidimensional.

Utilizînd regula de bază putem explica o serie de echivalențe dintre șir și pointer.

3.1.2. Echivalențe

Următorul program referă într-o manieră neobișnuită același element al șirul *a*.

Program 3.1.

```
/* ref.c - referință la șir */

main()
{
        static char a[]= "0123456789";
        int i=5;
        printf("%c\n", a[i]);
        printf("%c\n", i[a]);
}
```

Execuția programului afișează:

```
5
5
```

Prima referință la șir afișează al șaselea element, adică *5*. Următoarea ar părea o instrucțiune nepermisă, deoarece inversează variabila index *i* cu variabila șir *a*. Cu toate acestea programul este compilat fără erori și pe deasupra este și portabil.

Regula de bază explică dilema prin echivalența:

a[i] cu (*(a+i)) cu (*(i+a)) cu i[a]

Acest exemplu demonstrează că indexarea șirurilor este convertită întotdeuna la un pointer cu deplasament.

Folosind regula de bază putem obține alte expresii echivalente. De exemplu:

&a[i] cu &(*(a+i))

Ea a rezultat prin aplicarea operatorului *&* ambilor membri ai echivalenței. Precedența operatorilor *&* și * fiind de la stînga la dreapta, se pot elimina parantezele exterioare ale membrul drept și se obține:

&a[i] cu &*(a+i), care se reduce la:

&a[i] cu a+i

Programul următor demonstrează această echivalență.

Programul 3.2.

```
/* adrp.c - operatorii de adresare şi indirectare */

char a[] = "exemplu";
char *p=a;

main()
{
        /* utilizînd şirul */
        printf("%x %x\n", a+3, &*(a+3));
        /* utilizînd pointerul */
        printf("%x %x\n", p+3, &*(p+3));
}
```

Execuţia programului afişează:

```
179a 179a
179a 179a
```

Succesiunea *&** aplicată unui pointer cu deplasament are ca efect, la primul pas, selectarea conţinutului (valorii) de la respectiva adresă, iar la al doilea pas, selectarea adresei respectivului conţinut. Aşadar, aplicînd această succesiune unui obiect se obţine tot obiectul. Nu acelaşi lucru se poate spune despre succesiunea *&*. (a se vedea problema 1).

Echivalenţa:

&a[i] cu a+i

este folosită des în programele C. Fie declaraţiile:

```
#define MAX 100
int buf[MAX];
int *p;
```

Corespunzător acestor declaraţii, bucla:

```
for ( p=buf; p<&buf[MAX]; p++)
```

poate fi înlocuită cu bucla:

```
for ( p=buf; p<buf+MAX; p++)
```


Pentru ambele bucle compilatorul generează acelaşi cod în limbaj de asamblare.

O altă echivalenţă ce rezultă din regula de bază este posibilitatea referirii primului element al unui şir. Există echivalenţa:

a[0] cu (*(a+0)), ce se reduce la

a[0] cu *a

Rezultă că şirurile unidimensionale utilizează un singur nivel de indirectare.

Combinînd ultimele două echivalenţe rezultă echivalenţa:

&a[0] cu &*a, ce se reduce la
&a[0] cu a.

Argumentele funcţiilor utilizează frecvent echivalenţele:

a[0] cu *a şi
&a[0] cu a.

Un apel de forma *f(a[0])* este echivalent cu *f(*a)*, deoarece ambele expresii transmit valoarea primului element al şirului *a*. Similar, apelul *f(&a[0])* este echivalent cu *f(a)*. Se observă simplitatea formelor secunde.

3.1.3. Caracterele liniei de comandă

Argumentele liniei de comandă constituie un exemplu bun pentru a ilustra legătura existentă între indexarea şirurilor şi pointerii cu deplasament. Această legătură este vizibilă la accesul caracterelor liniei de comandă, după cum urmează:

Programul 3.3.

```
/* com.c - accesul la argumentele liniei de comandă */
#include <stdio.h>
main( int argc, char *argv[])
{
        if ( argc == 1) {
                printf("Utilizare: %s arg\n", argv[0]);
                exit(1);
        }
```

```
        printf("%c\n", *( argv[1] + 2));
        printf("%c\n", (*( argv + 1))[2]);
        printf("%c\n", argv[1][2]);
}
```

Programul **com.c** afișează pe trei linii consecutive al treilea caracter al primului argument (*argv[1]*). Fiecare argument al funcției *printf()* este diferit. Accesul la un caracter din *argv* se face fie utilizînd indexarea cuplată cu indirectare, fie două indexări. Expresiile sînt echivalente, deși la prima vedere ele nu par astfel. Se aplică regula de bază pentru a demonstra echivalența. Cea mai simplă expresie este *argv[1][2]*, care selectează al treilea caracter al șirului *argv[1]* folosind două indexări. Dacă facem substituția:

p=argv[1], atunci
argv[1][2] este echivalent cu p[2].

Aplicînd regula de bază, avem echivalența:

p[2] cu (*(p+2)), care după substituirea lui *p* devine:
argv[1][2] echivalent cu *(argv[1]+2)

unde am eliminat o pereche de paranteze din motive de redundanță. Expresia este mai clară decît cea anterioară, dcoarece se observă mai bine pointerul cu deplasament.

Aplicînd din nou regula de bază obținem echivalența:

argv[1] cu (*(argv+1)) și
argv[1][2] cu (*(argv+1))[2].

Perechea de paranteze exterioare nu se poate omite din cauza precedeței mai mari a indexării în raport cu indirectarea. Omiterea ei implică evaluarea expresiei **(argv+1)[2]* la primul caracter al argumentului al patrulea (*argv[3][0]* !!).

3.2. Expresii cu pointeri și șiruri

3.2.1. Scalari

Aritmetica cu pointeri este puternică, deoarece compilatorul este cel care calculează adresele. Expresiile cu pointeri scutesc programatorii de detaliile

de adresare ale memoriei. De exemplu, pentru a calcula un pointer cu deplasament, compilatorul utilizează dimensiunea obiectului referit, iar pentru a scădea doi pointeri la obiecte de același tip, compilatorul generează numărul de obiecte (elemente) aflate între cei doi pointeri. Acest număr este numit *scalar*.

Programele C utilizează scalarii în două situații: la calculul dimensiunii unor obiecte, cînd se folosește valoarea returnată de operatorul *sizeof*, și în aritmetica pointerilor cu scalari.

În aritmetica pointerilor cu scalari sînt permiși doi operatori aritmetici (+ și -). Dacă pointerul *p* referă un obiect din memorie și *n* este evaluat la un întreg, atunci expresiile:

```
p+n;
p-n;
```

reprezintă adrese corecte (valide) de memorie. Pentru a calcula adresa nouă, compilatorul nu utilizează valoarea variabilei *n*. Dacă ar utiliza-o, aritmetica pointerilor s-ar face pe octet. La calculul adresei, compilatorul va utiliza un scalar ce reprezintă dimensiunea obiectului.

Pentru a ilustra acest mecanism, să analizăm cîteva programe. Programul **depp.c** prezintă lucrul cu pointeri cu deplasament pe două șiruri, unul de tip *char* și altul de tip *float*. Programul afișează adresa de început a șirurilor și adresa celui de-al șaselea element din fiecare șir.

Programul 3.4.

```
/* depp.c - pointeri cu deplasament */

main()
{
        char cbuf[ 80], *p =cbuf;
        float fbuf[ 80], *q =fbuf;
        printf("cbuf = %u &cbuf[5]=p+5 =%u\n", cbuf, p+5);
        printf("fbuf = %u &fbuf[5]=q+5 =%u\n", fbuf, q+5);
}
```

Execuția programului afișează:

cbuf = 65098 &cbuf[5]=p+5 = 65103
fbuf = 65178 &fbuf[5]=q+5 = 65198

Se observă, din rezultatul execuției, că adresa elementului al șaselea este mai mare cu 5 octeți pentru șirul de tip *char* și cu 20 de octeți pentru șirul de tip *float*. Aceasta semnifică utilizarea de către compilator a scalarului 4 la calculul adresei corecte pentru *q+5*.

Următorul exemplu prezintă modul în care compilatorul calculează pointerul cu deplasament.

Program 3.5.

```
/* scalar1.c - operații cu pointeri și scalari */

main()
{
        char cbuf[ 80], *p =cbuf;
        float fbuf[ 80], *q =fbuf;
        printf("cbuf = %u &cbuf[5]=p+5 = %u\n", cbuf, p+5);
        printf("fbuf = %u &fbuf[5]=q+5 = %u\n", fbuf, q+5);
        printf("&cbuf[5] = %u\n",(char*)p + 5*sizeof( char));
        printf("&fbuf[5] = %u\n",(char*)q + 5*sizeof( float));
}
```

Execuția programului are ca rezultat:

cbuf = 65096 &cbuf[5]=p+5 = 65101
fbuf = 65176 &fbuf[5]=q+5 = 65196
&cbuf[5] = 65101
&fbuf[5] = 65196

Prima parte a execuției este similară programului anterior. Adresa celui de-al șaselea element se calculează în expresiile:

```
(char *)p + 5*sizeof(char);
(char *)q + 5*sizeof(float);
```

Aceste expresii explică modul de lucru al compilatorului. Scalarul este *sizeof(tip)*, adică *1* pentru tipul *char* și *4* pentru tipul *float*. Se utilizează castul pentru conversia pointerilor *p* și *q* la tipul pointer la *char*. Acest lucru este redundant pentru *p*, dar nu și pentru *q*. Fără acest cast programul nu calculează adresa corectă.

Generalizăm acest mod de calcul pentru orice tip de dată (șir, structură, uniune). Dacă *p* este un pointer la un obiect și *n* este o expresie întreagă, compilatorul calculează expresiile *p+n* ș

```
(char *)p + n*sizeof(tip_obiect);
(char *)p - n*sizeof(tip_obiect);
```

Scăderea pointerilor este posibilă doar dacă ei referă obiecte de același tip. Dacă *p* și *q* respectă această cerință, expresiile:

p-q și q-p

calculează numărul de elemente dintre cei doi pointeri. Rezultatul este un scalar tratat ca *int* sau *long*, funcție de implementare.

Următorul program declară un șir de 80 de caractere și inițializează doi pointeri pentru a referi două elemente din șir. Programul afișează numărul de caractere dintre cei doi pointeri.

Programul 3.6.

```
/* ncar.c - numărul elementor de tip char */

main()
{
        char buf[ 20], *q = buf, *p = q + 5;
        printf("adresa inferioară:%u ", q);
        printf("adresa superioară:%u\n", p);
        printf("diferența pointerilor este: %d\n",p-q);
}
```

Execuția programului afișează:

adresa inferioară:65478 adresa superioară:65483
diferența pointerilor este: 5

Programul următor realizează scăderea a doi pointeri, operație independentă de tipul pointerilor.

Programul 3.7.

```
/* ndouble.c - numărul elementelor de tip double */
```

```
main()
{
        double buf[ 20], *q = buf, *p = q + 5;
        printf("adresa inferioară:%u ", q);
        printf("adresa superioară:%u\n", p);
        printf("diferența pointerilor este: %d\n", p-q);
}
```

La execuția programului se afișează:

adresa inferioară:65336 adresa superioară:65376
diferența pointerilor este: 5

Șirul *buf* este de tip *double*, iar *p* și *q* sînt pointeri la acest șir. Diferența dintre adrese este de 40 de octeți, reprezentînd 5 elemente de tip *double*. Compilatorul împarte diferența dintre adrese la dimensiunea tipului unui element și obține numărul de elemente.

Programul **scalar2.c** prezintă modul în care compilatorul realizează scăderea a doi pointeri.

Programul 3.8.

```
/* scalar2.c - scăderea pointerilor */

main()
{
        double buf[ 20], *q = buf, *p = q + 5;
        printf("adresa inferioară: %u ", q);
        printf("adresa superioară:%u\n", p);
        printf("diferența scalară:%d\n",((unsigned int)p -
                (unsigned int)q) / sizeof( double));
        printf("diferența pointerilor este: %d\n", p-q);
}
```

Execuția programului are ca efect:

adresa inferioară:65336 adresa superioară:65376
diferența scalară: 5
diferența pointerilor este: 5

Programul calculează în două moduri diferența dintre cei doi pointeri. Utilizarea castului este necesară în expresia:

```
((unsigned int)p-(unsigned int) q)/sizeof(double)
```

entru ca scăderea să se facă în valoare absolută.

3.2.2. Expresii compacte cu pointeri.

Numim *expresii compacte cu pointeri* acele expresii cu pointeri ce utilizează operatorul de indirectare (*) și operatorii de incrementare (++) sau decrementare (--). Avantajul utilizării acestor expresii constă în reducerea dimensiunii programului și mărirea vitezei de execuție.

Fie *p* un pointer la un obiect. Tabelul 3.1 listează cele opt forme posibile de expresii compacte cu pointeri, separate în două grupe.

Expresie	Operatie	Afecteaz
*p++	postincrementare	pointerul
*p--	postdecrementare	pointerul
*++p	predecrementare	pointerul
*--p	predecrementare	pointerul
++*p	preincrementare	obiectul
--*p	predecrementare	obiectul
(*p)++	postincrementare	obiectul
(*p)--	postdecrementare	obiectul

Tabelul 3.1. Expresii compacte cu pointeri

Expresiile din prima grupă modifică pointerul și nu obiectul referit de acesta. Ele îmbunătățesc viteza de execuție a unui program și se recomandă utilizarea acestora, cu precădere în bucle.

Expresiile din a doua grupă afectează obiectul referit. Obiectele pot avea tipuri simple (int, char etc.) sau pot fi șiruri de tipuri simple. Compilatorul aplică operatorii ++ și -- obiectului referit și nu pointerului acestuia. Ca atare, nu putem utiliza în aceste expresii un pointer la o structură, uniune sau funcție.

Un obiect referit poate fi la rîndul lui un pointer. În acest caz, o expresie compactă din a doua grupă afectează un alt pointer. Acest lucru îmbunătățește viteza de execuție în bucle, asemenea pointerilor din prima grupă.

Să analizăm exemplele de expresii compacte cu pointeri din ambele grupe. Utilizăm pointeri de clasă de memorie register pentru a scurta timpul de evaluare a expresiilor. Expresiile:

```
*p++ și *p--
```

fac parte din prima grupă, iar evaluarea lor este guvernată de regula de precedență. Compilatorul evaluează obiectul referit de *p*, apoi incrementează (sau decrementează) pointerul. Această tehnică mută pointerul la următorul (respectiv anteriorul) obiect din memorie.

Următorul program utilizează expresii compacte cu pointeri de această formă. Presupunem existenta unui tablou ce conține un text demonstrativ, împărțit pe trei rînduri. Programul **prim.c** modifică tabloul pentru a tipări doar prima propoziție din fiecare rînd.

Programul 3.9.

```
/* prim.c - prima propoziție din tablou */

#include <stdio.h>

void prim( register char * []);

main()
{
        static char *linii[] = {
                "Prima propoziție. Ea conține: L 17 caractere.",
                "A doua propoziție. Ea conține: L 18 caractere.",
                "A treia propoziție. Ea conține: L 19 caractere.",
                (char *)NULL
        };
        int i;
        prim( linii);
        for( i=0; linii[ i]; i++)
                printf("%s\n", linii[ i] );
}
```



```
void prim( register char *plinie[])
{
        register char *pc;
        while ( pc = *plinie++ ) {
                while ( *pc++ != '.')
                        ;
                *pc = '\0';
        }
}
```

La execuție, programul afișează:

Prima propoziție.
A doua propoziție.
A treia propoziție.

Programul constă din două funcții: *main()* și *prim()*. Funcția *main()* conține declarația unui șir de pointeri la caracter ce referă cîte o linie din text. Ultimul pointer din șir este *NULL* și marchează sfîrșitul tabloului. Funcția principală apelează funcția *prim()*, avînd ca argument adresa tabloului. Ea caută primul caracter '.' din fiecare linie a textului și inserează după fiecare un caracter *NULL*, modificînd astfel linia textului inițial. După revenirea din funcția *prim()*, funcția *main()* va afișa liniile modificate ale tabloului.

Funcția *prim()* utilizează doi pointeri. Primul, *plinie*, este utilizat pentru a referi linia curentă din text. Al doilea, *pc*, referă caracterele din linia curentă. Figura 3.2 prezintă conținutul memoriei după prima execuție a instrucțiunii:

```
while( pc= *plinie++)
```

Această instrucțiune unifică trei operații. Dacă o descompunem obținem:

```
pc = *plinie;
plinie++;
while (pc);
```

Prima instrucțiune atribuie pointerului *pc* adresa liniei curente. A doua instrucțiune incrementează pointerul *plinie* pentru a referi următoarea linie (*linii[1]*), iar ultima face testul de sfîrșit de șir. Dacă *pc* este *NULL* se iese din buclă, în caz contrar se parcurge șirul de caractere prin **pc*.

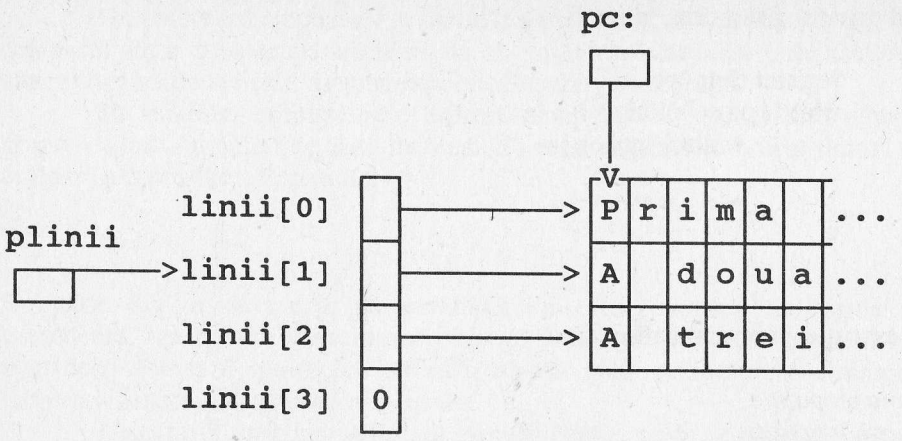


Fig. 3.2. while(pc=*plinie++);

Bucla internă conține următoarea expresie compactă:

*p++ != '.'

care determină parcurgerea șirului de caractere pîna la întîlnirea caracterului '.'.

Următoarele expresii cu pointeri:

*++p și *--p

determină incrementarea, respectiv decrementarea pointerului, înaintea evaluării obiectului referit. Efectul, în acest caz, este identic cu cel din cazul precedent, singura deosebire fiind ordinea de evaluare.

Programul **puncte.c** utilizează forma decrementată și afișează fiecare linie a aceluiași text pînă la caracterul ':'.

Programul 3.10.

```
/* puncte.c - pînă la caracterul ':' din fiecare linie */

#include <stdio.h>

void puncte( register char * []);
```



```
main()
{
        ... /* secvență de cod identică ca în programul 3.9 */
        puncte( linii);
        for( i=0; linii[ i]; i++)
                printf("%s\n", linii[ i] );
}
void puncte( register char *plinie[])
{
        register char *pc;
        while ( pc = *plinie++ ) {
                pc += strlen( pc);
                while ( *--pc != ':')
                        ;
                *pc = '\0';
        }
}
```

La execuție obținem:

Prima propoziție. Ea conține
A doua propoziție. Ea conține
A treia propoziție. Ea conține

Organizarea programului **puncte.c** este similară programului anterior. Deosebirea este dată de funcția *puncte()*, care pornește analiza șirului de la sfîrșitul liniei. La întîlnirea caracterului *':'* se face înlocuirea acestuia cu caracterul *'\0'*.

Pointerii au aceași semnificație ca în programul anterior. Analiza șirului pornește de la ultimul octet, poziționarea pe acesta fiind realizată de instrucțiunea:

pc += strlen(pc);

Expresia compactă: *--pc;

decrementează pointerul *pc* pentru a referi ultimul caracter al șirului.

Perechea de expresii din grupa a doua:

(*p)++ și (*p)--

afectează obiectul referit. Compilatorul incrementează, respectiv decrementează obiectul după evaluarea expresiei. Parantezele sînt necesare pentru stabilirea priorității.

Programul **secund.c** se referă la același tablou și afișează a doua propoziție din fiecare linie.

Programul 3.11.

```
/* secund.c - a doua propoziție din fiecare linie */

#include <stdio.h>

void secund( register char * []);

main()
{
        ... /* secvență de cod identică ca în programul 3.9 */
        secund( linii);
        for( i=0; linii[ i]; i++)
                printf("%s\n", linii[ i] );
}

void secund( register char *plinie[])
{
        do {
                while ( *(*plinie)++ != '.')
                        ;
                ( *plinie)++;
        } while ( *++plinie );
}
```

Execuția programului afișează:

Ea conține: L 17 caractere.
Ea conține: L 18 caractere.
Ea conține: L 19 caractere.

Figura 3.3. ilustrează conținutul memoriei după prima execuție a buclei:

```
while (*(*plinie)++ != '.')
```

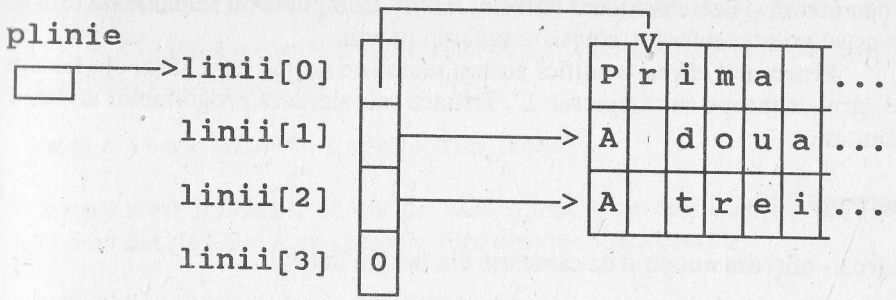



Figura 3.3. while (*(*plinie)++ != '.')

Expresia compactă cu pointer:

```
(*plinie)++
```

incrementează pointerul la caracter și nu pointerul la linie, iar

```
*(*plinie)++
```

se evaluează la un caracter din linia curentă, înainte de-a incrementa pointerul. După prima execuție a buclei *while*, codul modifică tabloul astfel încît *linii[0]* referă caracterul *'r'* (figura 3.3). În această manieră se analizează caracterele liniei curente pînă la întîlnirea caracterului '.', moment în care se părăsește bucla (*linii[0]* referă acum caracterul ' ').
Următoarea instrucțiune:

```
(*plinie)++;
```

mută pointerul la următorul octet, adică pe caracterul *'E'*. Expresia:

```
*++plinie
```

este o expresie compactă din prima grupă, ce realizează incrementarea pointerului la linie, atîta timp cît valoarea ei este diferită de *NULL*.

Ultima pereche de expresii:

```
++*p și --*p
```

incrementează şi decrementează obiectul referit. Compilatorul semnalează eroare dacă acest pointer referă structuri, uniuni sau funcţii.

Programul **cifre.c** modifică acelaşi tablou cu scopul de-a afişa, din fiecare linie, şirul ce începe cu caracterul *'L'*. Tehnica este similară programelor anterior prezentate.

Programul 3.12.

```
/* cifre.c - afişează numărul de caractere din fiecare linie */

#include <stdio.h>

void cifre( register char * []);

main()
{
        ... /* secvenţă de cod identică ca în programul 3.9 */
        cifre( linii);
        for( i=0; linii[ i]; i++)
                printf("%s\n", linii[ i] );
}

void cifre( register char *plinie[])
{
        do {
                *plinie += strlen(*plinie);
                while ( *(--*plinie) != 'L')
                                ;
        } while ( *++plinie);
}
```

Execuţia programului afişează:

L 17 caractere.
L 18 caractere.
L 19 caractere.

Expresiile compacte cu pointeri din aceste programe generează cod obiect mult mai scurt decît orice alte programe echivalente. Pentru a vă convinge de acest lucru rescrieţi funcţiile anterioare şi comparaţi dimensiunea codului generat şi eficienţa programelor.

3.2.3. Indexul negativ în tablouri

Combinăm regula de bază de la şiruri cu formula utilizată de compilator la pointeri cu deplasament:

&a[n] = a+n = (char *)a + n*sizeof(tip_obiect)

Compilatorul utilizează adresa de bază (numele şirului) cînd se accesează un element din şir. Dacă *n* este negativ, formula anterioară devine:

&a[-n] = a-n = (char *)a - n*sizeof(tip_obiect).

Rezultă aşadar că indexul negativ este permis în C.

Programul **limite.c** ilustrează folosirea indexului negativ, ceea ce semnifică depăşirea limitei din stînga sau dreapta a unui şir.

Programul 3.14.

```
/* limite.c - depăşirea limitei şirului */

main()
{
        int buf[ 10];
        buf[ -4] = 1;
        buf[ 10] = 2;
        printf("buf[-4]= %d iar buf[10]= %d", buf[-4], *(buf +10));
        printf(" Totul este corect ?\n");
}
```

Execuţia programului afişează:

buf[-4]= 1 iar buf[10]= 2 Totul este corect ?

Compilatorul nu face verificări de depăşire a marginilor şirului, motiv pentru care programul este compilat fără erori şi se execută corect. Acest program rescris in Pascal nu se va putea executa.

Acest lucru se poate exploata eficient în maniera ce urmează. Fie o versiune mai simplă a funcţiei de bibliotecă *calloc()*, pe care o numim *xcalloc()*. Această funcţie rezervă memorie heap pentru date şi verifică depăşirea spaţiului rezervat înainte de revenirea în funcţia apelantă.

Formatul de apel al funcţiei *xcalloc()* este identic cu cel al funcţiei *calloc()*. Funcţia *xcalloc()* alocă pe lîngă spaţiul cerut pentru date şi un spaţiu pentru un întreg. Funcţia memorează numărul de elemente pentru care s-a făcut alocarea în spaţiul întregului şi returnează adresa de început a spaţiului alocat. Programul utilizează indexul negativ la alocare pentru a putea accesa numărul de elemente.

Presupunem că apelăm funcţia *xcalloc()* pentru a aloca memorie pentru *10* întregi.

```
int *p;
...
p = (int *) xcalloc(10, sizeof(int));
...
```

Argumentele funcţiei sînt numărul de elemente şi dimensiunea în octeţi a fiecărui element. Utilizăm operatorul *sizeof* pentru portabilitate. Figura 3.4 ilustrează conţinutul memoriei după apelul funcţiei *xcalloc()*.

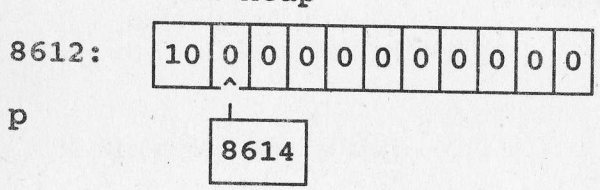


Fig. 3.4. Zonă alocată cu xcalloc() pentru 10 întregi

Pointerul *p* referă *10* întregi iniţializaţi cu zero. Numărul de elemente este memorat la adresa *8612*, adică înaintea zonei alocate pentru date.

Programul **neg.c** implementează funcţia *xcalloc()*:

Programul 3.14.

```
/* neg.c - index negativ */

#include <stdio.h>

void umple( int *);
void display( int *);
char *xcalloc( unsigned, unsigned);
```



```
main()
{
        char *xcalloc();
        int *p, *q;
        p = ( int *)xcalloc( 10, sizeof( int));
        q = ( int *)xcalloc( 15, sizeof( int));
        umple( p);
        display( p);
        umple( q);
        display( q);
}
```

Programul principal apelează de două ori funcţia *xcalloc()*. Primul apel rezervă o zonă pentru *10* întregi, iar al doilea pentru *15* întregi. Pointerii *p* şi *q* sînt corect alocaţi, deoarece funcţia testează în avans apariţia unei eventuale erori de alocare. Se apelează apoi funcţia *umple()*, pentru iniţializarea zonei cu valori întregi şi funcţia *display()*, pentru afişarea acestor valori. Cele două funcţii nu necesită ca argument numărul de elemente, acesta fiind accesibil prin pointerul *p*.

Codul funcţiei *xcalloc()* este:

```
char *xcalloc( unsigned nelem, unsigned dim)
{
        char *pm;
        char *malloc(), *memset();
        unsigned dimblc;
        dimblc = nelem + dim;
        if ( ( pm = malloc( dimblc +  sizeof( int))) == NULL) {
                printf("Eroare la alocare\n");
                exit(0);
        }
        *( int *) pm = nelem;
        memset( pm + sizeof( int), 0, dimblc);
        return pm + sizeof( int);
}
```

Variabila *dimblc* conţine dimensiunea zonei ce se alocă. Prin apelul funcţiei *malloc()* se alocă *dimblc* octeţi plus un întreg (doi octeţi). Dacă alocarea nu reuşeşte, se afişează un mesaj de eroare şi programul se termină. Instrucţiunea:

```
*(int *) pm = nelem;
```

memorează numărul de elemente în memorie. Utilizăm castul pentru a converti tipul pointerului la tipul *int* înainte de a memora dimensiunea zonei alocate. Apelul funcției *memset()* umple zona rezervată cu zero, pentru echivalarea funcției *xcalloc()* cu funcția *calloc()*. Instrucțiunea:

```
return ( pm+ sizeof(int));
```

returnează adresa de la care s-a făcut alocarea zonei de date.

Codul funcțiilor *umple()* și *display()*, care utilizează indexul negativ, este:

```
void umple( int *p)
{
        int i;
        int nelem = p[-1];
        for( i=0; i<nelem; i++)
                p[i]= i+1;
}

void display( int *p)
{
        int i;
        int nelem = p[-1];
        for ( i=0; i<nelem; i++)
                printf("%3d", p[i]);
        putchar('\n');
}
```

Ambele funcții accesează locația din memorie ce conține numărul de elemente, înainte de a face orice prelucrare asupra datelor. Instrucțiunea:

```
p[-1]
```

este similară cu **(p-1)* și se evaluează la un întreg ce reprezintă numărul de elemente.

Programul **neg.c** a demonstrat utilitatea indexului negativ doar pentru pointeri de tip întreg. La fel se utilizează *xcalloc()* pentru orice tip de dată. Apelul ei este:

```
struct orice *ps;
...
ps = (struct orice *) xcalloc( 20, sizeof( struct orice));
```


Acest lucru demonstrează independența funcției de tipul de dată utilizat. Deoarece numărul de elemente se memorează întotdeauna ca un întreg, este necesar castul pentru fiecare referire la numărul de elemente. Funcțiile de umplere și de afișare se modifică în aceeași manieră. Ca exemplificare, funcția de umplere devine:

```
sumple( struct orice *ps)
{
        int nelem = ((int *)p)[-1];
        ...
}
```

Parantezele sînt obligatorii, altfel se semnalează eroare la compilare. Aceeași modificare va apărea și în funcția *sdisplay()*.

3.3. Șiruri multidimensionale

Conceptele din acest paragraf constituie baza teoretică necesară înțelegerii tehnicilor ce urmează a fi prezentate. Luăm în discuție șirurile bi- și tridimensionale.

3.3.1. Șiruri bidimensionale.

Un șir bidimensional este, de fapt, un șir unidimensional ce are ca elemente șiruri unidimensionale. Lucrul cu aceste șiruri este ușor, dacă ele sînt privite ca tablouri împărțite în linii și coloane. Fiecare element al șirului este unic determinat de aceste două caracteristici.

Declarația:

```
double mint[3][5];
```

alocă memorie pentru *15* elemente de tip *double*. Șirul bidimensional *mint* conține *3* linii și *5* coloane (primul index indică linia, iar al doilea coloana). Indecșii liniei și coloanei încep de la valoarea zero. Prima linie este *mint[0]*, a doua este *mint[1]* ș.a.m.d.

Pentru a referi un element al tabloului bidimensional, este necesar formatul:

```
mint[linie][coloană]
```

Să considerăm un exemplu ce afișează un element al unui tablou bidimensional. Programul **tab.c** ilustrează o referire greșită la un element. Eroarea este frecventă la programatorii începători și nu este semnalată la compilare.

Programul 3.15.

```
/* tab.c - referirea elementelor dintr-un șir bidimensional */

main()
{
        static double mint[3][5]={
                { 1, 2, 3, 4, 5, },
                { 6, 7, 8, 9, 10,},
                { 11,12,13,14,15 }
        };
        printf("%g\n", mint[1][0]);
        printf("%g\n", mint[1,0]);
}
```

În urma execuției programului se afișează:

```
6
1.456798e-210
```

Primul apel *printf()* afișează primul element din linia a doua. Al doilea apel *printf()* ne-am aștepta să facă același lucru. Compilatorul evaluează primul index, după care acesta fiind urmat de operatorul ',', descarcă ultima evaluare și evaluează următorul index. Ca atare, este ca și cum am fi scris *mint[0]*. Acesta se evaluează la un pointer, deoarece *mint* este un șir bidimensional și se afișează valoarea flotantă a adresei *mint[0]*.

Programul **elem2.c** afișează adresele și dimensiunea elementelor unui șir bidimensional.

Programul 3.16.

```
/* elem2.c - elementele șirului bidimensional */

main()
{
        double mint[3][5];
```



```
        printf("mint mint[0] &mint[0][0]\n");
                /* adresa șirului */
        printf("%4x", mint);
                /* adresa primei linii */
        printf("%8x", mint[0]);
                /* adresa primului element */
        printf("%12x\n", &mint[0][0]);
                /* dimensiunea șirului */
        printf("%4d", sizeof(mint));
                /* dimensiunea unei linii */
        printf("%8d", sizeof(mint[0]));
                /* dimensiunea primului element */
        printf("%12d", sizeof(mint[0][0]));
}
```

Execuția programului afișează:

```
mint    mint[0] &mint[0][0]
ff62    ff62            ff62
 120     40               8
```

Adresele de memorie ale celor trei referințe la șir sînt aceleași. Operatorul *sizeof()* produce trei rezultate distincte. După cum ne așteptam, *mint[0][0]* este un element al șirului și, ca atare, are dimensiunea de *8* octeți (dimensiunea tipului *double*). Compilatorul tratează *mint* și *mint[0]* ca șiruri de dimensiuni diferite. Dimensiunea lui *mint* este de *120* octeți (de *15* ori dimensiunea tipului *double*), iar a lui *mint[0]* este de *40* de octeți (de *5* ori *double*).

La apariția unui nume de tablou într-o expresie, compilatorul utilizează pointeri. Fiecare pointer referă un obiect distinct. De exemplu, *mint[0]* este un pointer la primul din cele cinci elemente de tip *double* din prima linie, iar *mint* este un pointer la un șir de *5* elemente de tip *double*.

```
double mint[3][5]
```

Expresii ca acestea sînt frecvente în programe C. Tabelul 3.2. prezintă combinații de indirectări și indexări admise pe șiruri bidimensionale.

Tabelul nu prezintă toate posibilitățiile, dar din ele se pot obține la nevoie altele. De exemplu:

mint[0] este echivalentă cu *mint.

Expresie	Semnificatie
mint	pointer la sir de 5 double prima linie
**mint*	pointer la double, prima linie prima coloana
***mint*	double, valoarea din prima linie, prima coloana
mint[1]	pointer la double, linia a doua, prima coloana
**mint[1]*	double, valoarea din linia a doua prima coloana
mint+1	pointer la sir de 5 double, linia a doua
**mint+1*	pointer la double, prima linie, a doua coloana
mint[1][1]	double, valoare liniei si coloanei a doua
mint[1]+1	pointer la double, linia si coloana a doua

Talelul 3.2. Indirectări şi indexări pe şir bidimensional

În general, pentru a descifra o referinţă la un şir bidimensional se au în vedere următoarele:

1. O specificare completă a unei referinţe la şir reprezintă un element al şirului.
2. Orice specificare incompletă a unei referinţe se substituie cu un pointer.
3. Un nivel de indirectare este echivalent cu o indexare.

De exemplu:

mint[1][1]

este o valoare de tip *double*, deoarece referinţa e complet specificată. Referinţa:

mint[1]

este un pointer la *double*, pentru că este specificată incomplet. Similar:

mint

este un pointer la un pointer la *double*, care este echivalent cu pointer la un şir de *double*. Analog, pentru indirectare avem:

*mint

este un pointer la *double*, pentru că aplicăm indirectarea pe *mint*. Referinţa:

**mint

este o valoare *double*, deoarece sînt două nivele de indirectare. Expresia:

*mint[1]

este tot o valoare *double*, deoarece aplicăm indirectarea unui pointer.

3.3.2. Formula de acces pentru şirurile bidimensionale

Modelul de tablou cu linii şi coloane este util pentru ilustrarea unui şir bidimensional. Compilatorului îi rămîne realizarea corespondenţei dintre şir şi model. Din punct de vedere fizic memoria este accesată ca un şir unidimensional, aşa încît compilatorul reprezintă în memorie şirul bidimensional ca un bloc. Orice referire la un element din şir impune compilatorului calculul unei adrese în respectivul bloc.

Cunoaşterea modului în care compilatorul calculează această adresă explică de ce expresiile cu pointeri sînt mai rapide.

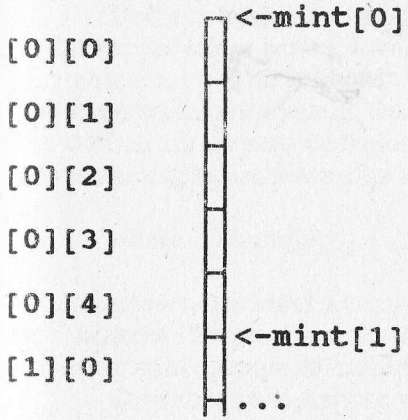


Figura 3.5. Blocul de memorie alocat pentru mint[3][5]

Compilatorul calculează adresa unui element al şirului prin localizarea liniei, iar apoi a coloanei din linie.

Pentru şirurile bidimensionale, expresiile:

```
mint[i][j]
*(mint[i] + j)
*(*(mint + i) + j)
```

sînt echivalente. Importantă este ultima. Din figura 3.2 rezultă că *mint* este un pointer la un şir de *5 double*. Atunci, *mint+i* mută pointerul cu *i* linii, în fiecare linie existînd *5* elemente de tip *double*. Ca atare compilatorul interpretează ultima expresie în maniera:

```
*(&mint[0][0] + 5*i +j)
```

Numim această expresie *formulă de acces*. Orice declaraţie de şir multidimensional implică o astfel de formulă.
Compilatorul utilizează această formulă pentru a translata indexarea:

```
mint[i][j]
```

la un pointer cu deplasament faţă de adresa de început a şirului.

Programul **map2.c** afişează adresele a două elemente din şirul bidimensional în două moduri distincte.

Programul 3.17.

```
/* map2.c - alocarea memoriei pentru şirului bidimensional */

main()
{
        double mint[3][5];
        int lin, col;
        double *adresa;

        lin=1;
        col=2;
                /* formula de acces */
        adresa= &mint[0][0] + 5*lin + col;
        printf("%u\n", &mint[1][2]);
```



```
                /* aceaşi adresa */
        printf("%u\n", adresa);
}
```

Execuţia programului afişează:

```
32778
32778
```

Adresele fiind aceleaşi, concluzia este clară.

Formula de acces pune în evidenţă şi alte aspecte. Se observă înmulţirea primului index cu numărul de coloane. Codul generat de compilator pentru această operaţie conţine o înmulţire, care de regulă consumă timp.

Un alt aspect semnificativ este redundanţa utilizării numărului de linii în declaraţie. Aceasta rezultă din rolul puţin important pe care îl are linia în formula de acces.

Un ultim aspect relevant este modul în care compilatorul realizează operaţia de înmulţire. Fie declaraţia:

```
double mint[3][4]
```

ca avînd formula de acces pentru *mint[i][j]*:

```
*(&mint[0][0] + 4*i +j)
```

Unele compilatoare generează în limbaj de asamblare instrucţiuni de deplasare spre stînga pentru a realiza operaţia de înmulţire. Înmulţind un întreg cu o putere a lui *2*, de exemplu, este echivalent cu a deplasa numărul respectiv spre stînga cu valoarea puterii. Dacă numărul de coloane nu este o putere de-a lui *2* se echivalează operaţia de înmulţire cu instrucţiuni de deplasare şi adunare. În programul **map2d.c** se putea afişa aceeaşi adresă şi cu instrucţiunea:

```
adresa = &mint[0][0] + (linie<<2)+linie + col;
```

Rezultă, aşadar, dependenţa formulei de acces de hard-ul calculatorului. Nu are sens scrierea de programe ce se bazează pe modul în care compilatorul implementează operaţia de înmulţire.

Concluzia ce se impune este evitarea lucrului cu şiruri bidimensionale prin utilizarea pointerilor.

3.3.3. Transmiterea şirurilor bidimensionale ca parametri pentru funcţii

Programul **extern2d.c** conţine declaraţia cu iniţializare:

```
double mint[3][5] = {
        { 1, 2, 3, 4, 5},
        { 6, 7, 8, 9,10},
        {11,12,13,14,15}
};
```

Programul **trans2d.c** transmite adresa acestui şir ca parametru funcţiei *trans2d()*, care afişează valoarea unui element din şir.

Programul 3.18.

```
/* trans2d.c -transmiterea şirului bidimensional
        ca argument în funcţie */

main()
{
        extern double mint[][5];
        int lin=2, col=1;

        printf("%g\n", mint[lin][col]);
        trans2d( mint, lin, col);
}

trans2d( double a[][5],int lin,int col)
{
        printf("%g\n", a[lin][col]);
}
```

Execuţia programului afişează:

```
12
12
```

Instrucţiunea:

```
        extern double mint[][5];
```


informează compilatorul că *mint* este un şir bidimensional de tip *double* declarat în alt fişier. Compilatorul nu alocă memorie. S-a omis primul index, în baza celor discutate anterior.

Limbajul C utilizează formula de acces pentru referirea şirurilor bidimensionale în interiorul funcţiilor care folosesc adresa şirului ca parametru.

În locul declaraţiei *double a[][5]* putem utiliza declaraţia *double (*a)[5]*, care arată mai clar ceea ce se transmite funcţiei. Compilatorul alocă memorie pentru un pointer la un şir de *5 double*. Parantezele sînt necesare, căci altfel am declara un şir de *5* pointeri la *double*. Referinţa la şiruri bidimensionale este un pointer cu deplasament de linie.

3.3.4. Şiruri tridimensionale

Şirurile tridimensionale adaugă o nouă dimensiune şirurilor bidimensionale. Un şir tridimensional este privit de către compilator ca un şir cu elemente şiruri bidimensionale. Din acest motiv, şirurile tridimensionale se vizualizează mai greu. Declaraţia:

```
        double mint[2][3][5];
```

declară şi alocă memorie pentru *30* de valori *double*, adică *240* de octeţi. Ne imaginăm acest şir ca format din două elemente, ce conţin fiecare *3* linii şi *5* coloane. Un element de acest gen îl numim **rînd**.

Referirea la un element al şirului tridimensional are formatul:

```
        mint[rînd][lin][col];
```

Extindem programele de la şirurile cu două dimensiuni la cele cu trei dimensiuni. Programul **elem3.c** afişează adrese şi dimensiuni dintr-un şir tridimensional.

Programul 3.19.

```
/* elem3.c - elementele tabloului tridimensional */
main()
{
        double mint[2][3][5];
        printf("mint mint[0] mint[0][0] &mint[0][0][0]\n");
                /* adresa şirului */
        printf("%4x", mint);
                /* adresa primului rînd */
```

```
        printf("%8x", mint[0]);
                /* adresa primei linii din rînd */
        printf("%12x", mint[0][0]);
                /* adresa primului element */
        printf("%16x\n", &mint[0][0][0]);
                /* dimensiunea şirului tridimensional */
        printf("%4d", sizeof(mint));
                /* dimensiunea unui rînd */
        printf("%8d", sizeof(mint[0]));
                /* dimensiunea unei linii */
        printf("%12d", sizeof(mint[0][0]));
                /* dimensiunea unui element */
        printf("%16d\n", sizeof(mint[0][0][0]));
}
```

Execuţia programului afişează:

mint	*mint[0]*	*mint[0][0]*	*&mint[0][0][0]*
feea	*feea*	*feea*	*feea*
240	*120*	*40*	*8*

Adresele sînt aceleaşi, dar operatorul *sizeof()* produce rezultate distincte. Pentru un şir tridimensional, *mint[0][0]* este o linie iar *mint[0]* este un rînd. Dimensiunea lor este de *40* de octeţi (*5 double* în fiecare linie), respectiv de *120* de octeţi (*15 double* în fiecare rînd).

Atragem atenţia asupra semnificaţiei actuale a referinţei:

```
mint[0][0]
```

care reprezintă un pointer la primul din cei *5 double*, iar:

```
mint[0]
```

este un pointer la prima din cele trei linii, adică este adresa primei linii din primul rînd. Pointerii conţin aceeaşi adresă, dar compilatorul atribuie semnificaţii distincte obiectelor referite.

Se permit aceleaşi expresii cu pointeri ca la şirurile bidimensionale, dar cu alte semnificaţii. Lor li se adaugă altele ca:

```
***mint         *mint[1][1]
**mint[1]       *mint[1] + 1
**mint+1        mint[1][1]+1.
```


3.3.5. Formula de acces pentru şiruri tridimensionale

Similar şirurilor bidimensionale, harta memoriei la alocare pentru un şir tridimensional apare tot sub forma unui bloc. Figura 3.8. ilustrează acest lucru:

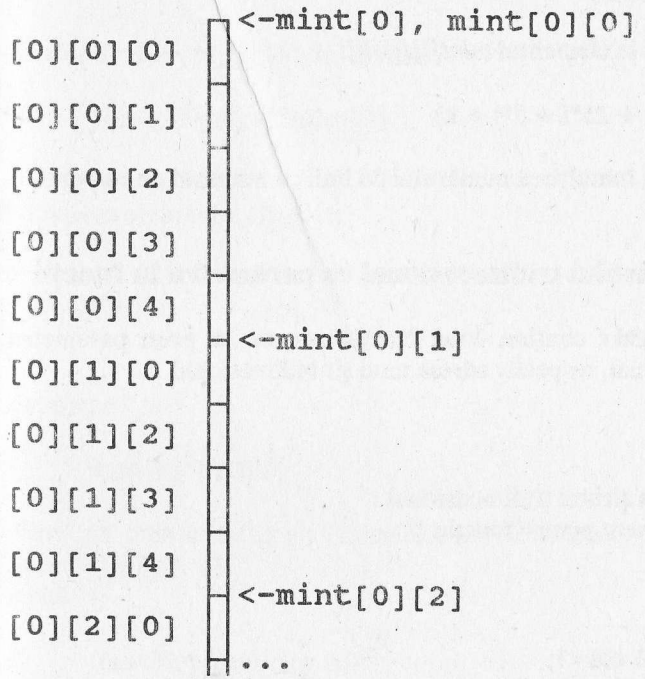


Figura 3.6. Blocul de memorie alocat pentru mint[2][3][5]

Următoarele linii:

```
mint[i][j][k]
*(mint[i][j] + k)
*(*(mint[i] + j) + k)
*(*(*(mint + i ) + j) + k)
```

sînt echivalente. Ultima expresie ilustrează modul în care compilatorul calculează adresa în bloc. Evaluarea începe cu *mint+i*, care este un pointer cu delasamentul de *15 double* înmulţit cu *i* faţă de începutul şirului. Expresia **(mint+i)+j* este un pointer cu deplasamentul de *5 double* înmulţit cu *j* faţă de adresa *mint + i*. În

final, expresia *(*(mint+i)+j)+k este un pointer cu deplasamentul de *k double* față de adresa anterioară. Astfel se obține adresa elementului *mint[i][j][k]*.

Translatînd cele discutate anterior pentru declarația:

```
double mint[2][3][5];
```

obținem formula de acces la elementul *mint[i][j][k]*:

```
*(&mint[0][0][0] + 15*i + 5*j + k)
```

Numărul *15* a rezultat din înmulțirea numărului de linii cu numărul de coloane.

3.3.6. Transmiterea șirului tridimensional ca parametru în funcții

Programul **trans3d.c** conține două funcții care au ca prim parametru adresa șirului tridimensional, respectiv adresa unui șir bidimesional.

Programul 3.20.

```
/* trans3d.c - transmiterea șirului tridimensional
                ca argument pentru funcție */

main()
{
        int rînd=1, lin=2, col=1;
        static double mint[2][3][5] = {
                {       { 1, 2, 3, 4, 5},
                        { 6, 7, 8, 9,10},
                        {11,12,13,14,15}
                },
                {
                        { 16,17,18,19,20},
                        { 21,22,23,24,25},
                        { 26,27,28,29,30}
                }
        };

        printf("%g\n", mint[rînd][lin][col]);
        trans3d( mint, rînd, lin, col);
}
```



```
trans3d( double (*b)[3][5], int rînd, int lin, int col)
{
        printf("%g\n", b[rînd][lin][col]);
        trans2d( b[rînd], lin, col);
}

trans2d( double (*a)[5], int lin, int col)
{
        printf("%g\n", a[lin][col]);
}
```

Execuția programului afișează:

```
27
27
27
```

Declarația:

```
double (*b)[3][5];
```

declară un pointer la un șir bidimensional de *15 double*. Ea este echivalentă cu *double b[][3][5]*.
Apelul:

```
trans2d( b[rînd], lin, col);
```

conține ca prim parametru pointerul la șirul bidimensional *b[rînd]*.

Rezultatul afișat demonstrează echivalența dintre cele trei posibilități de acces la un element dintr-un șir tridimensional.

3.3.7. Performanță și pointeri

După familiarizarea cu lucrul cu pointeri și șiruri multidimensionale putem să luăm în discuție problema vitezei de execuție a diverselor programe. Eliminăm din programele ce urmează formula de acces la elemente și substituim indexarea șirurilor prin expresii cu pointeri. Această măsură mărește viteza de execuție a unui program, indiferent de algoritmul pe care îl utilizează compilatorul pentru efectuarea operației de înmulțire.

Realizăm, în cele ce urmează, un calcul comparativ de performanță și ilustrăm avantajele declarării pointerilor la șiruri în locul utilizării șirurilor multidimensionale.

3.4. Expresii compacte cu pointeri în comparație cu șiruri indexate

Se consideră exemple de programe ce utilizează șiruri de tip întreg. Timpii obținuți au rezultat din execuția acestor programe pe configurațiile de calculatoare:

1. compatibil IBM 286 PC-AT 8-12 MHz, respectiv
2. compatibil IBM 386 PC-AT 16-21 MHz.

Pentru afișarea timpului se utilizează funcția de bibliotecă *gettime()* în maniera:

```
/* programul timp.c */

#include "dos.h"

timp()
{
        struct time d;
        gettime( &d);
        printf("ssec:%02d sec:%02d min:%02d\n",
                d.ti_hund, d.ti_sec, d.ti_min);
}
```

Primul program, **a1d.c**, declară un șir de tip întreg ale cărui elemente se inițializează de *10000* de ori.

Programul 3.21.

```
/* a1d.c - inițializarea șirului
        unidimensional folosind indexarea */

#include "timp.c"

#define IMAX 10
#define LOOP 10000
```



```
main()
{
        int a[IMAX];
        register int i;
        int v=0;
        timp();
        while ( v++ < LOOP)
                for(i=0; i<IMAX; i++)
        /* pointer cu deplasament */
                        a[i]=v;
        timp();
}
```

Am declarat variabila *i* de clasă de memorie *register* pentru ca programul să se execute cît mai rapid posibil.
Programul **p1d.c** realizează același lucru ca și programul anterior, dar utilizează expresia compactă cu pointer *(*p++)*.

Programul 3.22.

```
/* p1d.c - inițializarea șirului unidimensional
        folosind expresia compactă cu pointer */

#include "timp.c"

#define IMAX 10
#define LOOP 10000

main()
{
        int a[IMAX];
        register int *p;
        int v=0;
        timp();
        while ( v++ < LOOP)
                for(p=a; p<&a[IMAX];)
                        *p++=v;
        timp();
}
```

Timpii necesari execuţiei codului dintre apelurile funcţiei *timp()* (care conţine şi timpul necesar execuţiei funcţiei *timp()*) sînt:

1	2
a1d.c	
ssec:51 sec:48 min:05	ssec:30 sec:48 min:31
ssec:79 sec:48 min:05	ssec:39 sec:48 min:31
p1d.c	
ssec:18 sec:22 min:06	ssec:42 sec:53 min:07
ssec:40 sec:22 min:06	ssec:58 sec:53 min:07

Tabelul 3.3

Se observă execuţia mai rapidă a programului **p1d.c** pe configuraţia 1 de calculatoare, respectiv a programului **a1d.c** pe configuraţia 2 de calculatoare. Nici un program nu foloseşte formula de acces.

Programul **a2d.c** iniţializează un şir bidimensional în aceeaşi manieră.

Programul 3.23.

```
/* a2d.c - iniţializarea şirului bidimensional folosind indexarea */
#include "timp.c"
#define IMAX 10
#define JMAX 10
#define LOOP 30000
main()
{
        int a[IMAX][JMAX];
        register int j;
        register int i;
        int v=0;
        timp();
        while ( v++ < LOOP)
                for(i=0; i<IMAX; i++)
                        for(j=0; j<JMAX; j++)
                                a[i][j]=v;
        timp();
}
```


Deoarece indexul *j* este folosit mai des, declaraţia lui este prima (se ţine cont de observaţiile de la clasa de memorie *register*).
Compilatorul utilizează formula de acces pentru a accesa un element al unui şir bidimensional.

Programul **p2d.c** utilizează în locul indexării şirului o expresie compactă cu pointer. Este suficientă o singură buclă pentru a parcurge şirul.

Programul 3.24.

```
/* p2d.c - iniţializarea şirului bidimensional
        folosind expresia compactă cu pointer*/

#include "timp.c"

#define IMAX 10
#define JMAX 10
#define LOOP 30000

main()
{
        int a[IMAX][JMAX];
        register int *p;
        int v=0;
        timp();
        while ( v++ < LOOP)
                for(p=&a[0][0]; p<&a[IMAX][0];)
                        *p++=v;
        timp();
}
```

Programul iniţializează pointerul *p* cu adresa de început a şirului şi părăseşte bucla *for* cînd s-au iniţializat toate elementele şirului. Expresia:

```
p < &a[IMAX][0]
```

testează dacă pointerul *p* a accesat toate elementele şirului. Dacă ultima adresă validă din şir este:

```
&a[IMAX-1][JMAX-1]
```

prima adresă invalidă este *&a[IMAX][0]*.

Compilatorul nu utilizează formula de acces pentru selectarea unui element, deoarece indecşii şirului nu variază în program.

Expresiile:

p < a[IMAX] şi p < &a[IMAX][0]

sînt echivalente. Ele derivă din regula de bază conform succesiunii:

p < &a[IMAX][0]
p < &*(a[IMAX] + 0)
p < a[IMAX]

Execuţia programelor **a2d.c** şi **p2.c** afişează timpii:

1	2
a2d.c	
ssec:39 sec:42 min:31	ssec:30 sec:48 min:33
ssec:77 sec:48 min:31	ssec:43 sec:52 min:33
p2d.c	
sec:52 sec:51 min:33	ssec:12 sec:53 min:37
sec:45 sec:53 min:33	ssec:58 sec:54 min:37

Tabelul 3.4.

Rezultatul demonstrează că expresia compactă cu pointeri se execută mai rapid decît indexarea şirului, care foloseşte formula de acces.

Următoarea pereche de programe realizează acelaşi test pe şiruri tridimensionale.

Programul 3.25.

/* a3d.c - iniţializarea şirului tridimensional folosind indexarea */

```
#include "timp.c"

#define IMAX 10
#define JMAX 10
```



```
#define KMAX 10
#define LOOP 30000

main()
{
        int a[IMAX][JMAX][KMAX];
        register int k;
        register int j;
        register int i;
        int v=0;
        timp();
        while ( v++ < LOOP)
         for(i=0; i<IMAX; i++)
                for(j=0; j<JMAX; j++)
                        for(k=0; k<KMAX; k++)
                                a[i][j][k]=v;
        timp();
}
```

Programul 3.26.

/* p3d.c - iniţializarea şirului tridimensional
folosind expresia compactă cu pointer */

```
#include "timp.c"
#define IMAX 10
#define JMAX 10
#define KMAX 10
#define LOOP 30000

main()
{
        int a[IMAX][JMAX][KMAX];
        register int *p;
        int v=0;
        timp();
        while ( v++ < LOOP)
                for(p=&a[0][0][0]; p<&a[IMAX][0][0];)
                        *p++=v;
        timp();
}
```

Rezultatele execuției:

1	2
a3d	
ssec:66 sec:02 min:37	ssec:66 sec:12 min:39
ssec:04 sec:31 min:38	ssec:04 sec:42 min:39
p3d.c	
ssec:95 sec:29 min:40	ssec:35 sec:28 min:51
ssec:90 sec:48 min:40	ssec:20 sec:37 min:51

Tabelul 3.5.

Rezultatul este concludent.

3.4.1. Declarația pointerilor la șiruri.

O altă tehnică ce permite execuția mai rapidă a buclelor este bazată pe declarația care se utilizează. Surprinzător, tehnica este independentă de codul din interiorul buclei. În locul șirului multidimensional se declară un șir de pointeri. Șirul de pointeri elimină utilizarea formulei de acces și convertește indexarea în pointeri cu deplasament.

Fie un program care memorează într-un șir bidimensional cîteva nume de fructe și afișează apoi codul fiecărui fruct.

Programul 3.27.

```
/* ch2d.c - şir bidimensonal de caractere */

#include <ctype.h>      /* necesar funcției toupper() */

main()
{
        static char fructe[6][10] = {
                "mere", "alune", "cireșe",
                "gutuie", "portocale", "struguri"
        };
        int i;
```



```
        for(i=0; i<6; i++)
                printf("%c cod %s\n",
                        toupper(fructe[i][0]),fructe[i]);
}
```

Execuția programului afișează:

M cod mere
A cod alune
C cod cireșe
G cod gutuie
P cod portocale
S cod struguri

Numărul de coloane (10) în declarația tabloului *fructe* este numărul de caractere al celui mai lung șir (*"portocale"*), la care s-a adăugat octetul terminator de șir de caractere (*NULL*). Cum restul șirurilor sînt mai scurte, rezultă un număr de octeți neutilizați.

Codul afișat a rezultat din conversia primei litere a fructului (*fructe[i][0]*) în majusculă. Utilizăm un pointer la un fruct, *fructe[i]*, pentru a afișa numele fructului.

Acest exemplu pune în evidență cîteva dezavantaje.

Primul constă în alocarea ineficientă a memoriei pentru tabloul *fructe*, avînd în vedere lungimea diferită a numelui fructelor.

Al doilea se referă la cazul în care se adaugă la șirul fructelor inițiale un nume de fruct mai lung decît *"portocale"*. În acest caz trebuie mărit numărul de linii și coloane.

Al treilea dezavantaj implică utilizarea formulei de acces la referirea unui caracter din tablou.

O alternativă de eliminare a acestor dezavantaje constă în înlocuirea șirului bidimensional cu un șir de pointeri la caracter.

Programul următor este identic cu anteriorul, exceptînd declarația șirului *fructe*.

Programul 3.28.

```
/* ch1p.c - şir de pointeri la caracter */

#include <ctype.h>
```

```
main()
{
        static char *fructe[6] = {
                "mere", "alune", "cirese",
                "gutuie", "portocale", "struguri"
        };
        int i;
        for(i=0; i<6; i++)
                printf("%c cod %s\n",
                toupper(fructe[i][0]),fructe[i]);
}
```

Rezultatul execuției este identic.

Se elimină astfel octeții neutilizați din zona de date, prin utilizarea eficientă a memoriei. Există un șir de pointeri, fiecare dintre ei referind un nume de fruct. Dacă se adaugă un nume de fruct nou, este necesară memorie numai pentru numele fructului și pentru pointerul la el. Ultimul dezavantaj dispare și el dacă se observă echivalența expresiilor:

fructe[i] și *(fructe + i).

Ultima este un pointer cu deplasament, motiv pentru care compilatorul nu utilizează formula de acces. Similar, expresia:

fructe[i][0]

este echivalentă cu expresiile:

*fructe[i] și **(fructe + i).

Ultima este o dublă indirectare aplicată unui pointer cu deplasament.

Programul **ch1p.c** se va executa cu siguranță mai rapid decît **ch2d.c**.

Prezentăm în continuare și exemplele de programe ce lucrează pe șir tridimensional.

Programul 3.29.

```
/* ch3d.c - șir de pointeri la caracter */

#include <ctype.h>
```



```
main()
{
        static char fructe[2][6][10] = {
                {       "mere", "alune", "cireșe",
                        "gutuie", "pere", "struguri"
                },
                {       "banane", "cocos", "portocale",
                        "kiwi", "mango", "curmale"
                }
        };
        int i,j;
        for(i=0; i<2; i++) {
                printf( i==0 ? "indigene:\n" "tropicale:\n");
                        for(j=0; j<6; j++)
                                printf("\t%c cod %s\n",
                        toupper(fructe[i][j][0]),fructe[i][j];
        }
}
```

La execuție se afișează:

indigene:
M cod mere
A cod alune
C cod cireșe
G cod gutuie
P cod pere
S cod struguri
tropicale:
B cod banane
C cod cocos
P cod portocale
K cod kiwi
M cod mango
C cod curmale

Primul indice al tabloului *fructe* stabilește tipul fructelor, iar șirurile bidimensionale păstrează numele fructelor. Figura 3.7 ilustrează felul în care se memorează elementele șirului tridimensional.

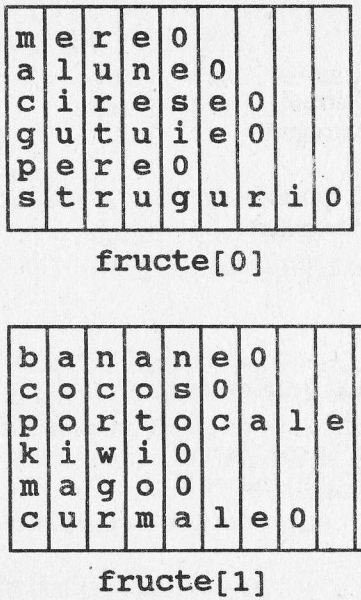


Figura 3.7. Şir tridimensional de caractere

Programul acesta conţine, la rîndul lui, dezavantajele programului **ch2d.c.**

Programul 3.30.

```
/* ch1pp.c - şir de pointeri la pointeri la caracter */
#include <ctype.h>
main()
{
        static char *indigene[6] = {
                "mere", "alune", "cirese",
                "gutuie", "pere", "struguri"
        };
        static char *tropicale[6] = {
                "banane", "cocos", "portocale",
                "kiwi", "mango", "curmale"
        };
        static char **fructe[2] = {
                indigene, tropicale};
        int i,j;
```



```
        for(i=0; i<2; i++) {
                printf( i==0 ? "indigene:\n" :"tropicale:\n");
                        for(j=0; j<6; j++)
                                printf("\t%c de la %s\n",
                        toupper(fructe[i][j][0]),fructe[i][j];
        }
}
```

Execuţia este identică cu cea a programului 3.29.
Am declarat şi iniţializat trei şiruri de pointeri. Primele două (*indigene* şi *tropicale*) conţin pointeri la şiruri constante, iar ultimul (*fructe*) conţine cei doi pointeri anteriori.

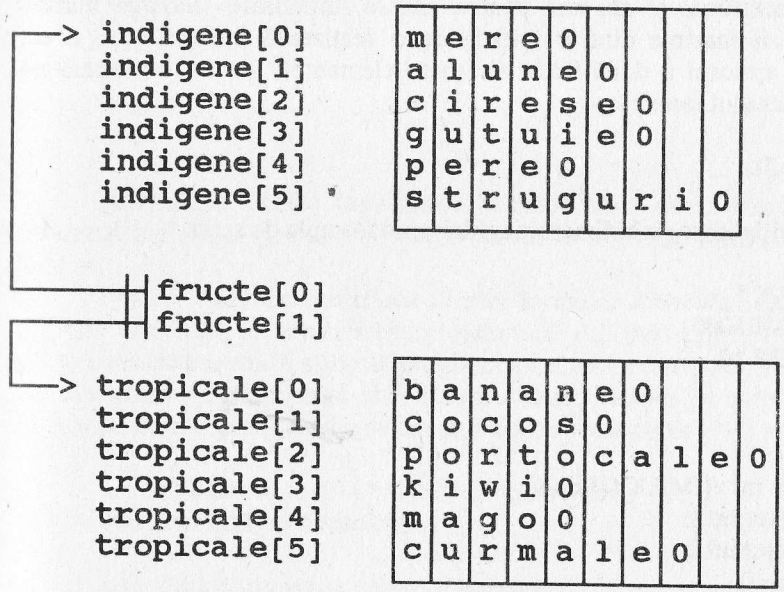


Figura 3.8. Şir de pointeri la pointer la caracter

Structura aceasta ierarhică cu pointeri este uşor modificabilă şi foarte eficientă din punctul de vedere al alocării memoriei.
Expresia:

```
fructe[i][j][0]
```

este echivalentă cu expresia *(*(*(fructe+i)+j)). Din acest motiv, compilatorul nu utilizează formula de acces.

3.4.2. Conversia şirurilor bidimensionale

Conversia şirurilor bidimensionale măreşte viteza de execuţie a unui program. Ea este utilă în sisteme soft mari, care conţin declaraţii de şiruri multidimensionale şi multe referiri la şiruri care trebuie convertite în expresii compacte cu pointeri. Testele făcute pe configuraţiile de calculatoare menţionate confirmă rezultatele modeste pentru conversii bidimensionale şi rezultatele substanţiale pentru conversii tridimensionale.

Programul următor, **sm2d.c**, iniţializează de zece ori cu numere aleatoare un şir bidimensional şi afişează pentru fiecare iniţializare valoarea maximă. Găsirea valorii maxime dintr-o iniţializare o realizează funcţia *maxv()*, care parcurge cu ajutorul a două bucle *for* toate elementele şirului bidimensional. Codul programului este:

Programul 3.31.

```
/* sm2d.c - iniţializare şir bidimensional folosind formula de acces */

#define IMAX 5
#define JMAX 5000
#define LOOP 10

main()
{
        static int a[IMAX][JMAX];
        register int j;
        register int i;
        int n=0;

        while ( n++ < LOOP) {
                for ( i=0; i<IMAX; i++)
                        for ( j=0; j<JMAX; j++)
                                a[i][j] = rand();
                printf("max = %d\n", maxv(a));
        }
}
```



```
maxv( int (*b)[JMAX])  /* echivalentă cu b[][JMAX] */
{
        register int j;
        register int i;
        int mv=b[0][0];  /* primul element */

        for ( i=0; i<IMAX; i++)
                for( j=0; j<JMAX; j++)
                        if ( b[i][j] > mv)
                                mv=b[i][j];
        return mv;
}
```

Execuţia programului afişează:

```
max = 32767
max = 32767
max = 32765
max = 32765
...
```

Am folosit pentru rapidizarea buclelor clasa de memorie *register* la declararea indecşilor şirului. În ambele funcţii se foloseşte formula de acces la element.

Pentru a elimina utilizarea acestei formule, schimbăm cîteva linii din program. O soluţie ar fi rescrierea programului utilizînd expresii compacte cu pointeri. Această metodă, aplicată programelor mari, necesită multe modificări în cod, care consumă timp şi pot introduce noi erori. Metoda pe care o prezentăm are avantajul modificării *minime* a codului. Ea respectă paşii:

1. Se redenumeşte şirul bidimensional original, păstrînd însă tipul datelor.

2. Se declară un şir de pointeri la acelaşi tip de dată, căruia i se dă numele şirului original. Lungimea acestui şir este egală cu numărul liniilor şirului original.

3. Se iniţializează şirul de pointeri cu adresele liniilor şirului original.

Programul modificat este **pi2d.c**, obţinut din **sm2d.c** prin metoda conversiei bidimensionale.

Programul 3.32.

```
/* pi2d.c - iniţializare şir bidimensional prin indirectare pointer */

#define IMAX 5
#define JMAX 5000
#define LOOP 10

main()
{
        static int ax[IMAX][JMAX];
        static int *a[IMAX] = {
                ax[0], ax[1], ax[2], ax[3], ax[4]
        };
        register int j;
        register int i;
        int n=0;
        while ( n++ < LOOP) {
                for ( i=0; i<IMAX; i++)
                        for ( j=0; j<JMAX; j++)
                                a[i][j] = rand();
                printf("max = %d\n", maxv(a));
        }
}

maxv( int *b[])   /* echivalent cu **b */
{
        register int j;
        register int i;
        int mv=b[0][0];  /* primul element */
        for ( i=0; i<IMAX; i++)
                for( j=0; j<JMAX; j++)
                        if ( b[i][j] >mv)
                                mv=b[i][j];
        return mv;
}
```

Figura de mai jos ilustrează zona de date pentru programul **pi2d.c**.

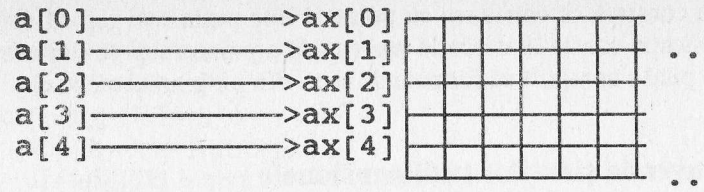


Fig.3.8. Conversia bidimensională

Pointerul *a[0]* conţine adresa primului element din prima linie. Similar, *a[1]* este un pointer la primul element din a doua linie, ş.a.m.d. Programul **pi2d.c** iniţializează elementele tabloului *a* cu valoarea zero. Această alternativă nu este recomandată, deoarece pentru şiruri mari ea nu este practică. Pentru a actualiza şirul de pointeri cu adresele liniilor se pot utiliza macrouri, dar cea mai bună alternativă rămîne scrierea unei funcţii (vezi problema 10).

Această tehnică nu modifică referirea unui element din şirul multidimensional. Ca atare, referirea din programul **pi2d.c** este identică cu cea din programul **smd2.c**, dar compilatorul utilizează indirectarea pointerului în locul formulei de acces.

În programul **sm2d.c** aveam în funcţia *maxv()* următoarea declaraţie:

```
int (*b)[JMAX], care este echivalentă cu
int b[][JMAX].
```

Variabila *b* era un pointer la un şir de *JMAX* întregi. În schimb, în programul **pi2d.c** declaraţia lui *b* este modificată la:

```
int *b[], care este echivalentă cu
int **b.
```

În acest caz, variabila *b* este un pointer la un pointer la întreg. Dacă nu se respectă aceste declaraţii, programele nu se vor executa corect.

Vitezele de execuţie pe două arhitecturi diferite sînt:

Program	Metoda	286(ssec)	386(ssec)
sm2d.c	sir	20.1	3.6
pi2d.c	pointer	19.3	3.0

Tabelul 3.6.

Se constată că versiunea cu pointeri este puțin mai rapidă decît cea cu șiruri. Explicația constă în execuția mai rapidă a pointerului cu deplasament din programul **pi2d.c** în raport cu formula de acces din programul **sm2d.c**.

3.4.3. Conversia șirurilor tridimensionale

Ultima pereche din setul de programe de test aplică tehnica de conversie la șirurile tridimensionale. Programul **sm3d.c** folosește un șir tridimensional în maniera programelor anterioare:

Programul 3.33.

```
/* sm3d.c - inițializare șir tridimensional folosind formula de acces */

#define IMAX 5
#define JMAX 5
#define KMAX 1000
#define LOOP 10

main()
{
    static int a[IMAX][JMAX][KMAX];
    register int k;
    register int j;
    register int i;
    int n=0;
    while ( n++ < LOOP) {
        for ( i=0; i<IMAX; i++)
            for ( j=0; j<JMAX; j++)
                for ( k=0; k<KMAX; k++)
                    a[i][j][k] = rand(); /* aleatoare */
        printf("max = %d\n", maxv(a));
    }
}

maxv( int (*b)[JMAX][KMAX])  /* b[][JMAX] */
{
    register int k;
    register int j;
    register int i;
```



```
    int mv=b[0][0][0]; /* primul element */
    for ( i=0; i<IMAX; i++)
        for( j=0; j<JMAX; j++)
            for ( k=0; k<KMAX; k++)
            if ( b[i][j][k] > mv)
                mv=b[i][j][k];
    return mv;
}
```

Cu execuția:

```
max = 32767
max = 32767
max = 32765
...
```

Declarația:

```
int (*b)[JMAX][KMAX]
```

declară variabila *b* ca pointer la un șir bidimensional. Programul **sm3d.c** utilizează formula de acces în funcția principală și în funcția *maxv()*.

Metoda pentru conversia șirurilor tridimensionale respectă pașii:

1. Se redenumește șirul tridimensional original, păstrîndu-se tipul datelor.
2. Se declară un șir bidimensional de pointeri la același tip de dată. Lungimea acestui șir este egală cu numărul de rînduri și linii ale șirului original.
3. Se inițializează șirul pointerilor cu adresele liniilor șirului original.
4. Se declară un șir de pointeri la pointeri de același tip.Lungimea acestui șir este egală cu numărul de rînduri ale șirului.
5. Se inițializează șirul de pointeri cu adresele liniilor șirului de pointeri din șirul anterior.

Conversia tridimensională pentru programul **sm3d.c** este realizată în programul **pi3d.c**:

Programul 3.34.

```
/* pi3d.c - inițializare șir tridimensional
          prin indirectarea pointerului */
```

```
#define IMAX 5
#define JMAX 5
#define KMAX 1000
#define LOOP 10

main()
{
        static int ax[IMAX][JMAX][KMAX];
        static int *ay[IMAX][JMAX] = {
                ax[0][0], ax[0][1], ax[0][2],
                        ax[0][3], ax[0][4],
                ax[1][0], ax[1][1], ax[1][2],
                        ax[1][3], ax[1][4],
                ax[2][0], ax[2][1], ax[2][2],
                        ax[2][3], ax[2][4],
                ax[3][0], ax[3][1], ax[3][2],
                        ax[3][3], ax[3][4],
                ax[4][0], ax[4][1], ax[4][2],
                        ax[4][3], ax[4][4],
        };
        static int **a[IMAX] = {
                ay[0], ay[1], ay[2], ay[3], ay[4]
        };
        register int k;
        register int j;
        register int i;
        int n=0;
        while ( n++ < LOOP) {
                for ( i=0; i<IMAX; i++)
                        for ( j=0; j<JMAX; j++)
                                for ( k=0; k<KMAX; k++)
                                a[i][j][k] = rand();
                printf("max = %d\n", maxv(a));
        }
}

maxv( int **b[]) /* echivalent ***b */
{
        register int k;
        register int j;
        register int i;
```



```
        int mv=b[0][0][0];          /* primul element */
        for ( i=0; i<IMAX; i++)
                for( j=0; j<JMAX; j++)
                        for ( k=0; k<KMAX; k++)
                        if ( b[i][j][k] > mv)
                                mv=b[i][j][k];
        return mv;
}
```

Execuția:

```
max = 32767
max = 32767
max = 32765
...
```

Figura 3.9 ilustrează noua organizare a datelor pentru programul pi3d.c.

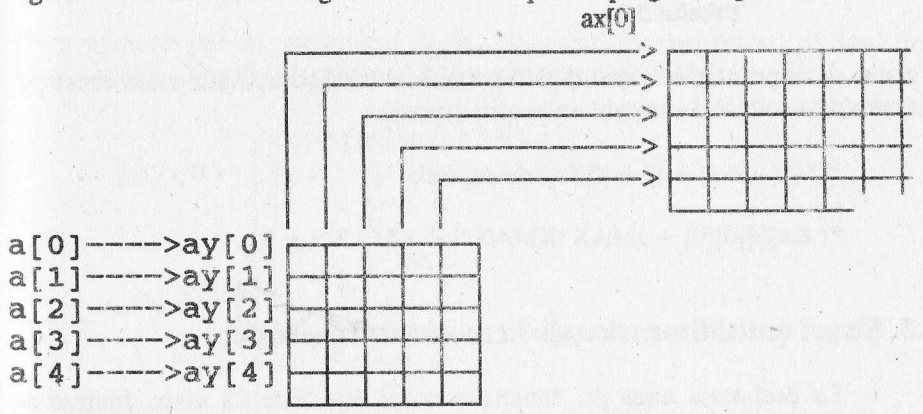


Figura 3.9. Conversia șirului tridimensional

Am prezentat doar pointerii de la *a* la *ay* și de la prima linie a lui *ay* (*ay[0]*) la primul rînd (*ax[0]*). *b[0]* este un pointer la prima linie a șirului bidimensional de pointeri la întregi. *ay[0][0]* este un pointer la primul întreg din prima linie, iar primul rînd (*ax[0][0][0]*), *ay[0][1]* este un pointer la primul întreg din a doua linie și primul rînd (*ax[0][1][0]*), ș.a.m.d. Șirul original (*a*) devine un șir de *IMAX* pointeri la pointeri la întregi. Se observă că necesarul suplimentar de memorie crește mult la metoda conversiei tridimensionale. Din aceleași motive ca și la conversia bidimensională, se recomandă scrierea unei

funcţii de iniţializare a şirului de pointeri şi nu iniţializarea lui cu valoarea zero (vezi problema 11).

Dacă transmitem adresa şirului tridimensional ca argument unei funcţii, trebuie modificată declaraţia din funcţia *maxv()*. Ea devine:

int **b[] echivalentă cu ***b

în care *b* este declarat pointer la un pointer la întreg.

Tabelul 3.7. prezintă viteza de execuţie a celor două programe:

Program	Metoda	286(ssec)	386(ssec)
sm3d.c	sir	30.3	4.3
pi3d.c	pointer	20.6	3.4

Tabelul 3.7.

Versiunea cu pointer este mai rapidă pe ambele configuraţii, din cauza execuţiei mai rapide a codului în asamblare pentru expresia:

((*(a + i) + j) + k) faţă de expresia

*(&a[0][0][0] + JMAX*KMAX*i + KMAX*j + k).

3.5. Şiruri multidimensionale în momentul execuţiei

La declaraţia unui şir, dimensiunea lui este fixă. Ca atare, înainte de execuţie se ia decizia asupra lungimii şirului, fără a ţine cont că el există pe stivă sau în zona de date. O soluţie care elimină dimensiunea anticipată a şirurilor este crearea şirurilor dinamice. Ca atare se alocă atîta memorie heap cît este nevoie. Se utilizează, în acest caz, funcţiile de alocare a memoriei heap.

Pentru şiruri unidimensionale este simplu, deoarece funcţia de alocare returnează adresa de început a zonei bloc alocate. Ea se poate referi ca şir. Se utilizează castul pentru pointerul returnat pentru a obţine un pointer la tipul de dată pentru care s-a făcut alocarea. Referirea la un element se face fie prin notaţia de şir, fie prin cea de pointer.

La şiruri multidimensionale problema este mai dificilă. Nu există funcţie de bibliotecă care să aloce memorie, astfel încît ea să poată fi utilizată ca un şir

multidimensional. Zona alocată este o zonă bloc formată din octeţi consecutivi, fără delimitare de rînd, linie sau coloană.

În continuare este prezentată o metodă de păstrare a conceptelor de rînd, linie şi coloană, astfel încît memoria heap alocată să pară programului un şir multidimensional. Prezentăm cîteva funcţii şi macrouri ce creează şiruri dinamice bi- şi tridimensionale. Aceste şiruri pot avea orice tip, putînd fi chiar şiruri de structuri sau uniuni. Metoda organizează memoria heap astfel încît referirea la şir este un pointer cu deplasament. Mai mult, programele pot utiliza notaţia şir cu rînd, linie şi coloană, ca şi cum am fi declarat şirul. În încheiere este prezentat un macrou ce permite funcţiilor C păstrarea convenţiei de apel la subrutinele din FORTRAN şi BASIC, care utilizează şiruri multidimensionale.

3.5.1. Tehnica bidimensională

Declaraţia:

int a[2][3];

alocă memorie pentru şase întregi. Zona bloc alocată este organizată pe două linii şi trei coloane. Figura 3.10 ilustrează dispunerea în memorie.

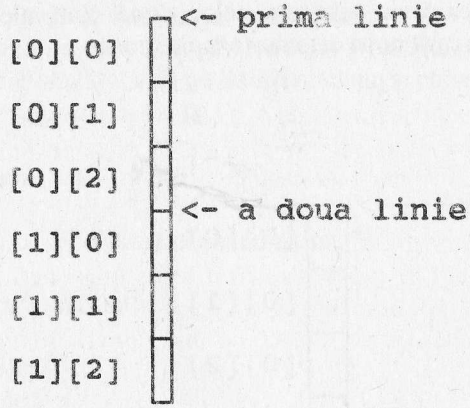


Fig. 3.10. Alocarea memoriei pentru şirul a[2][3]

În memorie, a doua linie urmează după prima. Compilatorul calculează adresa de memorie a unui element folosind formula de acces.

Pentru şirul anterior putem aloca memorie şi dinamic, utilizînd funcţia de bibliotecă *calloc()*. Utilizarea funcţiei *malloc()* nu iniţializează implicit elementele şirului cu valoarea zero. Instrucţiunile:

```
char *calloc();
int *p;
p = ( int *) calloc( 2*3, sizeof( int));
...
```

rezervă memorie heap conform figurii 3.10. Toate elementele sînt iniţializate cu valoarea zero. Referinţa la şir este un pointer cu deplasament, ce parcurge zona memoriei heap ca pe un şir unidimensional.

Putem vizualiza această zonă bloc ca un şir bidimensional, dar nu putem utiliza notaţia de şir bidimensional pentru referirea unui element. Cu alte cuvinte, *p[5]* este ultimul element din şir, dar formularea *p[1][2]* dă eroare la compilare. Din regula de bază rezultă echivalenţa:

p[1][2] cu *(p[1] + 2).

Dacă *p[1]* este evaluat la un întreg, compilatorul încearcă să dereferenţieze *p[1]+2*, ceea ce produce eroare la compilare.

Să presupunem că se alocă memorie heap adiţională pentru un şir de doi pointeri. Pointerii se actualizează cu adresele celor două linii ale şirului bidimensional. Figura 3.11. ilustrează noua dispunere în memorie:

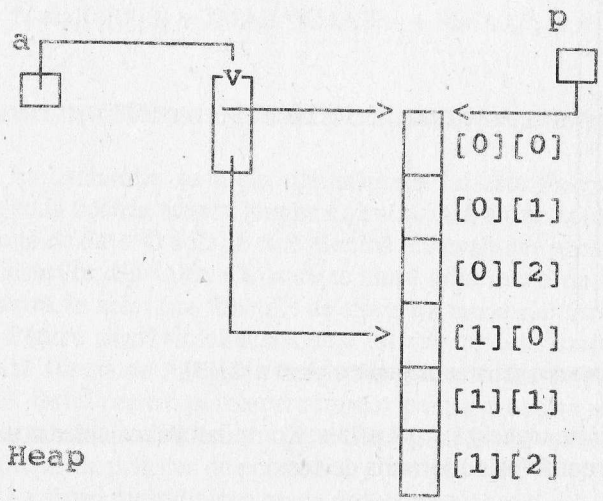


Figura 3.11. Alocare dinamică

Pointerul *p* conţine adresa de început a zonei alocate, iar *a* este adresa şirului de pointeri la linii. Instrucţiunile care iniţializează pointerii sînt:

```
int **a;
...
a = ( int **) calloc( 2, sizeof( int *));
a[0] = p;
a[1] = p + 3;
```

În acest mod putem referi datele ca şi cum ar fi elemente într-un şir bidimensional. Expresia *a[1][2]* este acum compilată şi referă ultimul element din şir. Referirea este echivalentă cu expresia **(a[1]+2)*, aşa încît compilatorul nu utilizează formula de acces.

Orice referinţă de tip şir utilizează pointerul *a* şi nu *p*. Aceste declaraţii permit utilizarea memoriei heap ca un şir bidimensional.

3.5.2. Funcţia de iniţializare pentru şirul bidimensional

Prezentăm o funcţie ce iniţializează şirurile dinamice ilustrate în figura 3.11. Programul următor necesită două argumente în linia de comandă. Primul este numărul de linii, iar al doilea numărul de coloane ale şirului bidimensional. Programul creează în memoria heap şirul bidimensional, îl iniţializează cu date, afişează elementele lui pe linii şi coloane şi eliberează spaţiul alocat la terminare. Codul programului este:

Programul 3.35.

```
/* idim2.c - şir bidimensional dinamic */

#include <stdio.h>

int **idim2();
void ifree2();

main( int argc, char *argv[])
{
        int **a;
        int lin, col;
        int inum = 1;
        register int i, j;
```

```
    if ( argc != 3) {
        fprintf( stderr, "Utilizare: %s lin col\n", argv[0]);
        exit(0);
    }
    lin = atoi( argv[1]);
    col = atoi( argv[2]);
    a = idim2( lin, col);
    for (i=0; i<lin; i++)
        for ( j=0; j<col; j++)
            a[i][j]=inum++;
    for (i=0; i<lin; i++)
        for ( j=0; j<col; j++)
            printf("%5d%c", a[i][j], (j+1) % col ? ' ' : '\n');
    ifree2( a);
}
```

Execuția lui pentru argumentele liniei de comandă: *3 6* este:

```
 1    2    3    4    5    6
 7    8    9   10   11   12
13   14   15   16   17   18
```

Pointerul *a* este adresa șirul bidimensional creat, iar *idim2()* este funcția ce alocă memorie heap pentru acest șir. Funcția returnează un pointer la un pointer la întreg, care se atribuie apoi pointerului *a*. Programul utilizează notația cu doi indici la inițializarea șirului *a* și la afișarea valorilor elementelor sale.

Funcția *ifree2()* eliberează zonele alocate pentru șir, utilizînd pointerul *a* ca argument. Datele de test demonstrează corectitudinea celor discutate.
La execuția cu argumentele *5* și *9* se afișează:

```
 1    2    3    4    5    6    7    8    9
10   11   12   13   14   15   16   17   18
19   20   21   22   23   24   25   26   27
28   29   30   31   32   33   34   35   36
37   38   39   40   41   42   43   44   45

 1.1  2.1  3.1  4.1  5.1  6.1  7.1  8.1  9.1
10.1 11.1 12.1 13.1 14.1 15.1 16.1 17.1 18.1
19.1 20.1 21.1 22.1 23.1 24.1 25.1 26.1 27.1
28.1 29.1 30.1 31.1 32.1 33.1 34.1 35.1 36.1
37.1 38.1 39.1 40.1 41.1 42.1 43.1 44.1 45.1
```


Codul funcțiilor *idim()* și *ifree2()* este:

```
int **idim2( int lin, int col)
{
    char *calloc();
    int i;
    register int **plin, *pdata;
    pdata = ( int *) calloc( lin*col,
            sizeof(int));
    if ( pdata == ( int *)NULL) {
        fprintf( stderr, "Nu există
                spațiu\n");
        exit(1);
    }
    plin = ( int **) calloc( lin, sizeof( int *));
    if ( plin == ( int **)NULL) {
        fprintf( stderr, "Nu există spațiu\n");
        exit(1);
    }
    for ( i=0; i<lin; i++) {
        plin[ i] = pdata;
        pdata += col;
    }
    return plin;
}

void ifree2( int **pa)
{
    void free();
    free( *pa);
    free( pa);
}
```

Pointerii *plin* și *pdata* sînt similari pointerilor *a* și *p* din figura 3.10. Instrucțiunea:

```
pdata = (int *) calloc( lin*col, sizeof(int));
```

apelează funcția *calloc()* pentru a aloca memorie heap de *lin*col* elemente de tip întreg.
Utilizăm castul (*int* *) pentru a converti tipul pointerului *pdata* de la caracter la întreg. Similar, instrucțiunea:

```
plin = ( int **) calloc( lin, sizeof( int *));
```

apelează funcția *calloc()* pentru a aloca memorie heap pentru un șir de pointeri. Lungimea șirului este de *lin* elemente de tip pointer la întreg.

Bucla *for* servește pentru inițializarea șirului de pointeri cu adresele primelor date din fiecare linie. Instrucțiunea:

```
plin[i] = pdata;
```

inițializează fiecare element al șirului referit de *plin* cu valoarea curentă a pointerului *pdata*. Instrucțiunea:

```
pdata += col;
```

utilizează aritmetica cu pointeri pentru a muta pointerul *pdata* la linia următoare. La părăsirea buclei toți pointerii au fost inițializați.
Figura 3.12. ilustrează aceasta:

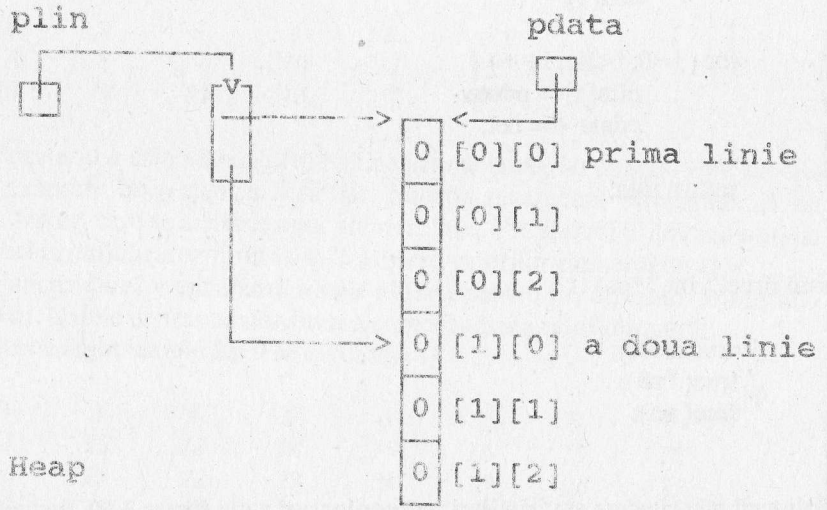


Figura 3.12. Alocare dinamică prin idim2()

Funcția *ifree2()* eliberează cele două zone alocate anterior prin apelurile funcției calloc(). Acesteia îi este suficientă transmiterea ca argument a adresei șirului de pointeri. Expresia **pa* referă zona alocată pentru date, iar pointerul *pa*

referă zona alocată pentru pointerii la linii. Cele două apeluri ale funcției de bibliotecă *free()* eliberează efectiv cele două zone de memorie.

3.5.3. Macrou pentru șir bidimensional

Funcția *idim2()* creează doar șiruri bidimensionale de tip întreg. Pentru a crea un șir bidimensional de orice tip, compilatorul trebuie să cunoască tipul șirului în momentul compilării. Acest lucru nu este dificil de realizat, existînd în acest sens cîteva metode. Una dintre ele ar fi modificarea funcției *idim2()* astfel încît ea să returneze un pointer la caracter, care apoi să fie convertit printr-un cast la tipul dorit. Programele apelează funcția *idim2()* cu operatorul *sizeof()* ca parametru, pentru a se specifica funcției dimensiunea unui element al șirului bidimensional.

O altă metodă ar consta în scrierea unui macrou. Avantajul ar fi viteză la execuție, deoarece macrourile se execută mai rapid decît funcțiile, iar dezavantajul constă în creșterea dimensiunii codului. Compromisul între viteza execuției și lungimea codului se face funcție de aplicație.

Programul **mdim2.c** este similar lui **idim2.c**, dar utilizează un macrou pentru crearea celor două șiruri. Fișierul **mdim.h** conține macrourile *DIM2* și *D2FREE*. Programul declară doi pointeri *a* și *b* pentru șirurile bidimensionale de întregi, respectiv *double*. Programul **mdim2.c** utilizează macroul *DIM2* pentru crearea ambelor șiruri. Programul inițializează primul șir cu valori întregi, iar al doilea cu valori reale. Primele cinci linii afișate la execuție conțin valorile întregi, iar următoarele cinci valorile reale.

Primul apel macro *DIM2* creează șirul de întregi. Apelul:

```
DIM2( a, lin, col, int);
```

are patru parametri. Primul parametru este un pointer la un pointer la întreg. El este inițializat în macrou și permite referirea unui element din șirul bidimensional prin notația, cu doi indici. Al doilea și al treilea parametru sînt numărul de linii, respectiv numărul de coloane. Ultimul parametru este tipul unui element al șirului bidimensional (în acest caz *int*).

Instrucțiunea:

```
DIM2( b, lin, col, double);
```

utilizează același macrou pentru a crea un șir bidimensional de tip *double*. După ce programul a afișat elementele șirului, se utilizează macroul *D2FREE* pentru a

elibera zona memoriei heap alocată pentru șir. Parametrul macroului *D2FREE* este pointerul pe care *DIM2* l-a inițializat cu adresa zonei bloc alocate.

Programul 3.36.

```
/* mdim2.c - macroul pentru șir dinamic bidimensional */

#include <stdio.h>
#include "mdim.h"

main( int argc, char *argv[])
{
        int **a;
        double **b;
        int lin, col;
        register int i, j;
        int inum = 1;
        double dnum = 1.1;
        if ( argc != 3) {
                fprintf( stderr, "Utilizare: %s lin col\n",
                        argv[0]);
                exit(1);
        }
        lin = atoi( argv[1]);
        col = atoi( argv[2]);
        DIM2( a, lin , col, int);
        i=0;
        for ( i=0; i<lin; i++)
                for ( j=0; j<col; j++)
                        a[i][j] = inum++;
        for ( i=0; i<lin; i++)
                for ( j=0; j<col; j++)
                        printf("%5d%c", a[i][j], (j+1) % col ? ' ' : '\n');
        putchar('\n');
        DIM2( b, lin, col, double);
        for ( i=0; i<lin; i++)
                for ( j=0; j<col; j++)
                        b[i][j] = dnum++;
        for ( i=0; i<lin; i++)
                for ( j=0; j<col; j++)
                        printf("%5g%c", b[i][j], (j+1) % col ? ' ' : '\n');
```



```
        D2FREE(a);
        D2FREE(b);
}
```

Fișierul **mdim.h** conține codul pentru macrourile *DIM2* și *D2FREE*.

```
/* mdim.h - macrourile */

#define DIM2( plin, lin, col, tip) \
{\
        void *calloc();\
        register tip *pdata;\
        int i;\
        pdata = (tip *) calloc( lin*col, sizeof( tip));\
        if ( pdata == (tip *)NULL) {\
                fprintf( stderr, "Eroare alocare spațiu
                        pentru date\n");\
                exit(1);\
        }\
        plin = (tip **) calloc( lin, sizeof( tip *));\
        if ( plin == (tip **)NULL) {\
                fprintf( stderr, "Eroare alocare spațiu
                        pentru șirul pointerilor la linii\n");\
                exit(1);\
        }\
        for (i=0; i<lin; i++) {\
                plin[i] = pdata;\
                pdata += col;\
        }\
}

#define D2FREE( plin) \
{\
        void free();\
        free(*plin);\
        free( plin);\
}
```

S-a rescris o bună parte din codul funcției *idim2()*. Macroul *DIM2* conține parametrul *tip* care apare în declarația pointerului *pdata* și în cast. În

momentul compilării, preprocesorul substituie textual parametrul *tip* cu *int* la primul apel și cu *double* la al doilea apel al macroului *DIM2*.

Se constată utilizarea diferită a parametrului *plin* în macrou față de funcție. Dacă funcția *idim2()* returna pointerul *plin* programului apelant, macroul *DIM2* substituie cuvîntul *plin* cu numele unui șir. La prima utilizare a lui *DIM2* el se substituie cu *a*, iar la a doua utilizare se substituie cu *b*. În acest mod macroul inițializează șirul de pointeri cu adresele de linii din memoria heap.

A nu se uita blocul care delimitează macroul, deoarece acesta face invizibile declarațiile de variabile cu același nume din programul principal.

Macroul *D2FREE* este similar funcției *ifree2()*.

Macrourile definite în această secțiune admit șiruri dinamice bidimensionale de orice tip (structuri, uniuni, pointeri, etc). Fie o structură numită *orice*. Pentru alocarea și eliberarea spațiului pentru un șir bidimensional de *3x4* elemente de tipul structurii de mai sus, utilizăm instrucțiunile:

```
#include "mdim.h"
struct orice **c;
...
DIM2( c, 3, 4, struct orice);
...
D2FREE(c);
...
```

3.5.4. Tehnica tridimensională

Avantajul utilizării tehnicii bidimensionale la șirurile dinamice este materializat prin utilizarea șirului de pointeri din memoria heap. Această abordare determină compilatorul să utilizeze indirectarea pointerului în locul formulei de acces. Dacă se extinde această tehnică la șiruri tridimensionale, avantajul este mai mare, deoarece referirea elementelor unui șir tridimensional devine mult mai rapidă. Pentru a realiza acest lucru trebuie definite șiruri de pointeri care să permită utilizarea memoriei heap alocate ca șir tridimensional. Singurul dezavantaj este creșterea memoriei alocate pentru șirurile de pointeri.

Declarația:

```
int a[2][3][2];
```

alocă memorie pentru douăsprezece elemente de tip întreg. Ele sînt organizate ca două șiruri bidimensionale de trei linii și două coloane. Figura de mai jos ilustrează dispunerea șirului în memorie.

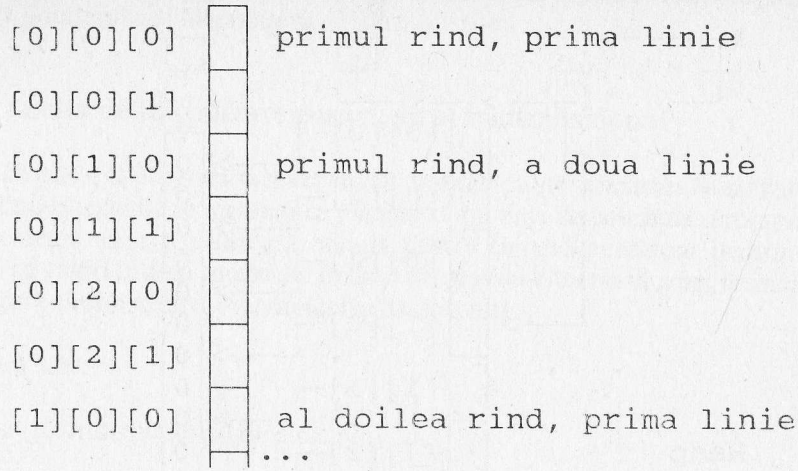


Figura 3.13. Alocare statică pentru a[2][3][2]

Al doilea rînd urmează primului, iar compilatorul va utiliza formula de acces pentru adresarea unui element.

Dacă șirul este alocat dinamic, instrucțiunile sînt:

```
char *calloc();
int *p;
...
p = (int *) calloc( 2*3*2, sizeof(int));
...
```

Locațiile de memorie alocate sînt succesive, dar se pierd noțiunile de rînd, linie și coloană. La acest nivel nu se poate utiliza notația pentru șir tridimensional, adică notația cu trei indici (notația tridimensională). Referințe ca:

```
p[0][2][1];
```

nu se vor compila. Ca atare, este nevoie de memorie heap adițională pentru a adresa corect datele.

Se alocă memoria heap pentru două șiruri de pointeri. Primul conține pointeri la linii, iar al doilea pointeri la rînduri.

Figura 3.12 ilustrează noua dispunere în memorie.

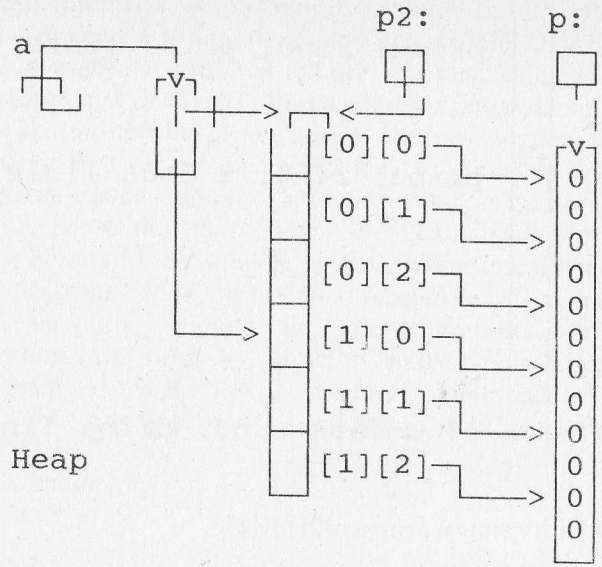


Figura 3.14. Alocare dinamică pentru a[2][3][2]

Pointerii *a*, *p2* și *p* pot exista în zona stivă, în zona de date, în zona heap sau în regiștrii interni ai calculatorului. Șirurile se află în zona heap. Dimensiunea șirului referit de pointerul *a* este egală cu numărul de rînduri. Fiecare element al acestui șir referă o adresă din șirul al doilea de pointeri. Această adresă conține adresa liniilor din fiecare rînd. Din acest motiv, dimensiunea șirului de pointeri, referită de *p2*, este egală cu produsul dintre numărul rîndurilor și numărul coloanelor. Cu acestă dispunere, referința la șir:

a[0][2][1] devine

*(a[0][2] + 1)

sau *(*(a[0] + 2) + 1)

Pentru a accesa zona memoriei heap cu notația tridimensională, compilatorul utilizează indirectarea și pointeri cu deplasament.

Declarațiile pentru cei trei pointeri sînt:

int ***a, **p2, *p;

Referirea la șir folosește pointeri cu triplă indirectare. Notația tridimensională utilizează pointerul *a* și nu pointerii *p2* sau *p*.

3.5.5. Funcția de inițializare pentru șirul tridimensional

Următorul program creează un șir tridimensional dinamic. Numărul de rînduri, linii și coloane se transmit ca parametri din linia de comandă. Programul **idim3.c** creează șirul și utilizează notația pentru șir tridimensional pentru a-l inițializa cu valori întregi succesive. În final programul eliberează zona memoriei heap alocată șirului. Codul programului principal este:

Programul 3.37.

```
/* idim3.c - șir tridimensional dinamic */

#include <stdio.h>

int ***idim3();
void ifree3();

main( int argc, char *argv[])
{
        int ***a;
        int rînd, lin, col;
        register int i, j, k;
        int inum = 1;
        if ( argc != 4) {
                fprintf(stderr,"Utilizare: %s lin col\n",
                        argv[0]);
                exit(1);
        }
        rînd = atoi( argv[1]);
        lin = atoi( argv[2]);
        col = atoi( argv[3]);
        a = idim3( rînd, lin, col);
        for ( i=0; i<rînd; i++)
                for ( j=0; j<lin; j++)
                        for ( k=0; k<col; k++)
                                a[i][j][k] = inum++;
```

```
        for ( i=0; i<rînd; i++) {
                printf("Rînd %d:\n", i);
                for ( j=0; j<lin; j++)
                        for ( k=0; k<col; k++)
                                printf("%5d%c", a[i][j][k], (k+1) % col
                                        ?' ' : '\n');
        }
        ifree3(a);
}
```

Execuția programului pentru argumentele liniei de comandă *2, 4, 8*, este:

Rînd 0:

1	*2*	*3*	*4*	*5*	*6*	*7*	*8*
9	*10*	*11*	*12*	*13*	*14*	*15*	*16*
17	*18*	*19*	*20*	*21*	*22*	*23*	*24*
25	*26*	*27*	*28*	*29*	*30*	*31*	*32*

Rînd 1:

33	*34*	*35*	*36*	*37*	*38*	*39*	*40*
41	*42*	*43*	*44*	*45*	*46*	*47*	*48*
49	*50*	*51*	*52*	*53*	*54*	*55*	*56*
57	*58*	*59*	*60*	*61*	*62*	*63*	*64*

Funcția *idim3()* creează șirul tridimensional. Programul memorează valoarea returnată de această funcție într-un pointer *a*, care va putea fi utilizat în notația de șir tridimensional. Execuția programului afișează elementele șirului pe linii și coloane pentru fiecare rînd. Funcția *ifree3()* utilizează pointerul *a* ca argument pentru a putea elibera spațiul alocat.

Codul pentru funcțiile *idim3()* și *ifree3()* este:

```
int ***
idim3( int rînd, int lin, int col)
{
        char *calloc();
        int i;
        register int ***prînd, **plin, *pdata;
        pdata = (int *) calloc( rînd*lin*col,sizeof(int));
        if ( pdata == (int *) NULL) {
                fprintf( stderr,"Nu există spațiu pentru date\n");
                exit(1);
        }
```



```
        plin=(int **) calloc(rînd*lin, sizeof(int *));
        if ( plin == (int **) NULL) {
                fprintf( stderr,"Nu există spațiu pentru
                        pointerii la linii\n");
                exit(1);
        }
        prînd=(int ***) calloc( rînd, sizeof(int **));
        if ( prînd == (int ***) NULL) {
                fprintf( stderr,"Nu există spațiu pentru
                        pointerii la rînd\n");
                exit(1);
        }
        for ( i=0; i<rînd*lin; i++) {
                plin[i] = pdata;
                pdata += col;
        }
        for ( i=0; i<rînd; i++) {
                prînd[i] = plin;
                plin += lin;
        }
        return prînd;
}

void ifree3( int ***pa)
{
        void free();

        free( **pa);
        free( *pa);
        free( pa);
}
```

Funcția *idim3()* utilizează trei pointeri. Primul, *prînd*, referă șirul pointerilor la rînd, al doilea, *plin*, referă șirul pointerilor la linii, iar ultimul, *pdata* referă datele. Toate șirurile se găsesc în zona heap. Instrucțiunile:

```
        pdata=(int *) calloc( rînd*lin*col,sizeof(int));
        plin=(int **) calloc( rînd*lin, sizeof(int **));
        prînd=(int ***) calloc( rînd, sizeof(int *));
```

alocă spațiu în memoria heap și inițializează pointerii sus-menționați; utilizează castul pentru conversia tipului pointerului returnat de funcția *calloc()*. Primul apel al funcției *calloc()* alocă memorie heap pentru date. Numărul de elemente pentru care se face alocarea îl reprezintă produsul dintre numărul de rînduri, linii și coloane. Tipul unui element este *int*, iar zona alocată se inițializează cu zero. Următoarele apeluri ale funcției *calloc()* alocă memorie heap pentru șirurile de pointeri. Dimensiunea primul șir este rezultatul produsului dintre numărului de rînduri și numărul de linii, iar dimensiunea celui de-al doilea șir este egală cu numărul de rînduri.

Funcția *idim3()* conține două bucle *for* pentru a inițializa șirurile de pointeri cu adresele elementelor șirului. Prima buclă este similară celei din programul **idim2.c**. Numărul de elemente este produsul *rînd*lin*, întrucît pointerii trebuie inițializați la fiecare linie din fiecare rînd. A doua buclă inițializează elementele șirului de pointeri la rînd cu adresa primei linii din șirul de pointeri anterior inițializați. Pentru a fi corecte, cele două bucle *for* trebuie scrise neapărat în această ordine. Funcția returnează pointerul *prînd* programului principal.

Funcția *ifree3()* este similară funcției *ifree2()*. Se apelează funcția *free()* de trei ori pentru a elibera cele trei zone alocate din memoria heap. Se eliberează zona alocată pentru date, cea pentru pointerii la linii și cea pentru pointerii la rînd. Respectarea acestei ordini este obligatorie.

Funcția *idim3()* creează doar șirul tridimensional dinamic. Funcția și macroul care creează și eliberează un șir tridimensional de orice tip sînt subiectul problemelor 14 și 15. Tehnica se poate extinde și la șiruri cu mai mult de trei dimensiuni.

3.5.6. Convenția la apelul subrutinelor

Limbajul de progamare FORTRAN permite transmiterea șirurilor multidimensionale ca parametri în funcții. Ca atare, acesta permite utilizarea notației pentru șir (cu doi sau trei indici) la referirea la un element. În FORTRAN, instrucțiunile:

```
SUBROUTINE FUNC( A, M, N)
DIMENSION A( M,N)
...
END
```

permit funcției *FUNC()* să acceseze orice șir bidimensional *A*, de *M* linii și *N* coloane. Această funcție poate fi apelată pentru șiruri de dimensiuni diferite. Biblioteca FORTRAN utilizează subrutine ca *FUNC()* pentru a inversa o

matrice, pentru a calcula un determinant sau pentru a determina valorile proprii etc. Convenția este utilă, deoarece subrutinele pot accesa elementele unui șir prin doi indici (de linie și coloană).

Limbajul C nu permite acest lucru. Iată un exemplu ce pune în evidență aceast neajuns. Fie declarația:

```
int data[5][9];
```

ce definește un șir bidimensional de *5x9* întregi. Presupunem existența unei funcții *func()*, care acceptă ca parametru șirul bidimensional. Codul ei ar putea fi:

```
func( int a[][9], int lin, int col)
{
        register int i, j;
        for ( i=0; i<lin; i++)
                for ( j=0; j<col; j++)
        /* referire corectă la șir */
                ...a[i][j]...
}
```

Apelul:

```
func( data, 5, 9);
```

impune cifra *9* ca necesară în declarația șirului *a*, pentru a putea folosi formula de acces. Dacă însă, se declară un alt șir prin:

```
int data1[4][10];
```

apelul:

```
func( data1, 4, 10);
```

nu este corect, deoarece funcția *func()* este compilată cu numărul de coloane inițializat la *9* și nu la *10*. Acest lucru determină accesul incorect la memorie.

O alternativă ar fi transmiterea adresei șirului ca parametru în funcție. Instrucțiunile:

```
func(&data[0][0], 5, 9);
func(&data[0][0], 4, 10);
```

permit funcției *func()* să acceseze șirul bidimensional ca un șir unidimensional. Codul funcției *func()* se modifică, după cum urmează:

```
func( int *a, int lin, int col)
{
        register int *p=a, *end = a + lin*col;
        while ( p<end) {
                /* nu pot utiliza notația de șir a[i][j] */
                ...*p++...
        }
}
```

Parametrul *a* este un pointer la întreg (anterior era un pointer la șir de întregi). Acest lucru permite utilizarea expresiei compacte cu pointeri pentru a accesa succesiv zona alocată pentru șir. Lungimea zonei alocate este de *lin*col* elemente întregi. Accesul la element este mai rapid, dar dispar noțiunile de linie și coloană. Dar o funcție ce calculează determinantul unei matrici are nevoie de aceste informații.

Programul următor utilizează un apel de macrou pentru a imita convenția de apel corespunzătoare șirurilor bidimensionale din FORTRAN. Programul **det.c** declară două șiruri bidimensionale de tip *double*. Primul are patru linii și patru coloane, iar al doilea cinci linii și cinci coloane. Șirurile materializează două matrici pătrate de ordin patru, respectiv cinci. Adresa și dimensiunea șirului se transmit unei funcții *det()*, care calculează determinatul. Nu interesează algoritmul de calcul al determinantului, ci tehnica aplicată funcției, deoarece ea necesită acces bidimensional la șir. Codul programului este:

Programul 3.38.

```
/* det.c - calculul determinantului */
#include <stdio.h>
double det();

main()
{
        static double f[4][4] = {
                1, 3, 2, 1,
                4, 6, 1, 2,
                2, 1, 2, 3,
                1, 2, 4, 1
        };
```



```
        static double g[5][5] = {
                1, 3, 2, 1, 7,
                4, 6, 1, 2, 6,
                2, 1, 2, 3, 5,
                1, 2, 4, 1, 4,
                8, 5, 4, 1, 3
        };
        printf("Determinantul lui f = %g\n",
                        det(f,4));
        printf("Determinantul lui g = %g\n",
                        det(g,5));
}

double det( double *arg, int n)
{
        register int i, j, k;
        double **a;
        double ret;
        double x;
        /* creare dinamică a șirului */
        DIMENSION2( arg, a, n, n, double);
        for ( k=0; k<n-1; k++)
                for ( i=k+1; i<n; i++) {
                        x = a[i][k]/ a[k][k];
                        for ( j=k; j<n; j++)
                                a[i][j] = a[i][j] - x*a[k][j];
                }
        for ( ret=1, i=0; i<n; i++)
                ret *= a[i][i];
        free(a);
        return ret;
}
```

La execuție programul afișează:

Determinantul lui f = 43
Determinantul lui g = 906

Șirurile bidimensionale *f* și *g* au dimensiuni diferite. Programul transmite adresele lor unei funcții *det()*, care calculează determinanții și afișează valorile acestora.

În blocul funcției *det()* se poate utiliza notația pentru șir bidimensional, deoarece macroul *DIMENSION2* permite acest lucru. Acest macrou este similar cu *DIM2*, cu o singură excepție. Întrucît datele există, singurul lucru pe care macroul îl face este inițializarea șirului de pointeri cu adresele de linii. Din acest motiv macroul necesită adresa șirului bidimensional (*arg*) ca prim parametru. Parametrul al doilea este un pointer la un pointer la *double* (*a*) pe care îl utilizăm ca șir bidimensional. Parametrii trei și patru sînt numărul de linii respectiv coloane, iar ultimul parametru este tipul datei.

Apelului de macrou îi urmează calculul determinantului, eliberarea memoriei heap alocate și returnarea valorii determinantului.

Definirea macroului *DIMENSION2* este:

```
#define DIMENSION2( arga, usera, lin, col, tip)\
        { \
        int i;\
        usera=(tip **) calloc( lin, sizeof( tip *));\
        if ( usera == (tip **) NULL) {\
                fprintf(stderr, "Nu există spatiu liber\n");\
                exit(1);\
        }\
        for ( i=0; i<lin; i++)\
                usera[i] = (tip *)((char *) arga + \
                        ((sizeof(tip)*i*col))); \
        }
```

Se apelează funcția *calloc()* pentru a aloca spațiu în memoria heap pentru șirul de pointeri la linii. Se utilizează bucla *for* pentru inițializarea acestor pointeri, *usera[i]*, cu adrese de linii. Adresa de început a datelor este dată de pointerul *arga*. Se utilizează castul pentru conversia tipului pointerului și se calculează pointerul cu deplasament pentru fiecare început de linie. Variabila *i* din interior blocului macroului nu este accesibilă funcției *det()*.

Tehnica se aplică cu succes și șirurilor tridimensionale. Macroul *DIMENSION3* este subiectul problemei 16.

În concluzie, macroul și funcția de inițializare a șirurilor dinamice bidimensionale și tridimensionale (nu și cele din programul 3.38) oferă, în funcții, posibilitatea accesului la un element cu notația de șir. Aceste programe se bucură și de avantajul execuției mai rapide a codului, avînd în vedere utilizarea indirectării pointerilor în locul formulei de acces. Pentru a demonstra acest lucru apelăm funcția *idim2()* pentru crearea a două șiruri de dimensiune diferită și transmitem pe rînd adresele lor aceleași funcții *func()*.
Iată codul:


```
int **a;
int **b;
...
a = idim2( 3, 4);
b = idim2( 5, 6);
...
func( a, 3, 4);
func( b, 5, 6);
...
```

Definirea funcției *func()* este:

```
func( int **c, int lin, int col)
{
        register int i, j;
        for ( i=0; i<lin; i++)
                for ( j=0; j<col; j++)
                        ...c[i][j]...
}
```

Funcția poate utiliza referirea la un element prin notația bidimensională de șir, dacă șirul a fost creat cu funcția *idim2()*. Parametrul *c* trebuie declarat pointer la un pointer la întreg. Aceeași abordare este valabilă dacă se utilizează macroul *DIM2*.

Dacă utilizăm șiruri tridimensionale create cu funcția *idim3()*, declarația de pointer este:

```
int ***c;
```

3.6. Probleme

1. Compilați următorul program:

```
/* a2.c - ilustrează operatori de adresă și indirectare */
char a[] = "incorect";
char *p = a;
main()
{
        printf("%x %x\n", a+3, *&(a+3));
        printf("%x %x\n", p+3, *&(p+3));
}
```

Explicați mesajul compilatorului. De ce operatorii *& se evaluează altfel decît &*?.

2. Studiați programul și interpretați rezultatul execuției.

```
/* comc2.c - accesul la caracterele liniei de comandă */
#include <stdio.h>
main( int argc, char *argv[])
{
        if ( argc == 1) {
                fprintf(stderr,"Utilizare: %s arg\n",argv[0]);
                exit(1);
        }
        printf("%c\n", argv[0][1]);
        printf("%c\n", *(*argv + 1));
        printf("%c\n", (*(argv))[1]);
        printf("%c\n", argv[1][0]);
        printf("%c\n", *argv[1]);
        printf("%c\n", *(*(argv + 1)));
}
```

Demonstrați echivalența expresiilor din cele două grupe de cîte trei instrucțiuni de tipărire.

3. Se consideră declarațiile:

```
char buf[80], *q = buf, *p = q + 5;
```

Care dintre expresiile de mai jos generează erori la compilare?

```
buf[p-q];
p += p-q;
q + p - q;
p - q + p;
q - p - q;
```

4. Se consideră programul:

```
/* pointarg - modificarea tabelei pentru a referi argumentele comenzii */
#include <stdio.h>
#define NRARG 3
main()
{
        static char *tabl[NRARG] = {
                "dir | more",
                "fox aplic",
                "xcopy *.doc a:*.txt"
        };
        int i;
        pointarg( NRARG, tabl);
        for ( i=0; i<NRARG; i++)
                printf("%s\n", tabl[i]);
}

pointarg( int dim, char *intr)
{
...
}
```

Se cere să se scrie funcția *pointarg()*, astfel încît execuția programului să afișeze:

```
| more
aplic
*.doc a:*.txt
```

Modificarea pointerilor în tabel se va face prin utilizarea de expresii compacte cu pointeri.

5. Programele de la expresii compacte cu pointeri lucrează cu un șir de pointeri la caracter. Două dintre programe apelează funcția *strlen()* pentru a determina lungimea fiecărei linii. Să se rescrie programul utilizînd un șir bidimensional de caractere. Acest lucru elimină utilizarea funcției *strlen()*, întrucît dimensiunea fiecărei linii devine fixă. Această abordare rapidizează programul ?. Dacă este posibil să se testeze viteza la execuție și să se compare rezultatele.

6. Se consideră declarația:

```
double mint[2][3][5];
```

Ce reprezintă referirile:

```
*mint
**mint
mint[1]
```

```
***mint
*mint[1]
mint + 1
**mint[1]
**mint + 1
*mint[1][1]
mint[1][1] + 1
```

7. Să se modifice programul **map2.c** pentu un șir tridimensional (programul **map3d.c**). Se utilizează formula de acces pentru afișarea adreselor ?. Cum se utilizează formula de acces pentru a afișa valoarea unui element pornind de la adresa lui ?.

8. Presupunem declarațiile:

```
int șir[23][80];
int buf[23][80][8];
int lin = 15, col = 50, pixel = 4;
```

Care dintre referirile la șir determină compilatorul să genereze instrucțiuni în asamblare de înmulțire sau deplasare-adunare ?. Să se utilizeze formula de acces pentru a găsi adresa calculată de compilator.

```
șir[10]
șir[lin]
șir[10][40]
șir[lin][40]
șir[10][col]
buf[15]
buf[15][50]
buf[lin][col]
buf[4][col][8]
buf[lin][50][4]
buf[lin][col][3]
buf[15][50][pixel]
```

9. Să se rescrie programele **sm2d.c** și **sm3d.c** utilizînd expresii compacte cu pointeri. Să se execute aceste programe și să se compare timpii lor de execuție. Cum este acest timp în comparație cu cel al programelor *pi2d.c* și *pi3d.c*?.

10. Presupunem că un program conține declarația:

```
static long șir[20][30];
```

Utilizînd tehnica de conversie a șirului bidimensional să se completeze codul:

```
static long xșir[20][30];
static long *șir[20];
...
load2d( xșir, șir, 20);
...
```

Funcția *load2()* încarcă fiecare element al celui de-al doilea argument cu adresele liniilor șirului bidimensional, ce apare ca prim argument. Să se scrie un program care apelează funcția *load2()* și testează rezultatele. În această abordare șirul de pointeri trebuie să fie declarat static ?.

11. Se consideră programul ce conține declarația:

```
static long buf[20][30][10];
```

Utilizînd tehnica de conversie a șirului tridimensional, să se completeze codul:

```
static long xbuf[20][30][10];
static long *ybuf[20][30];
static long **buf[20];
...
load3d( xbuf, ybuf, buf, 20, 30);
...
```

Funcția *load3d()* are două acțiuni:
a) încarcă fiecare element al celui de-al doilea argument cu adresele liniilor și coloanelor șirului tridimensional, ce apare ca prim argument.
b) încarcă al treilea argument cu pointerii la șirul bidimensional transmis ca al doilea argument. Să se scrie un program ce apelează *load3d()* și să se testeze rezultatele.

12. Să se rescrie functiile *load2d()* și *load3d()* pentru a utiliza memoria heap în locul zonei de date. Ambele funcții alocă spațiu pentru șirul de pointeri și îi inițializează cu adresele corespunzătoare.

13. Să se modifice funcția *idim2()* pentru a crea un șir bidimensional de orice tip. Următorul cod ilustrează apelul funcției *idim2()* pentru a crea două șiruri bidimensionale, unul de tip întreg de dimensiune *3x5* iar altul de tip *double* de dimensiune *4x6*.

```
...
char **idim2();
int **a;
double **b;
a = (int **) idim2( 3, 5, sizeof(int));
b = (double **) idim2( 4, 4, sizeof(double));
...
```

Versiunea trebuie să conțină mesaje de diagnosticare a erorilor.

14. Să se scrie macroul pentru crearea unui șir tridimensional de un tip oarecare.

```
#define DIM3( prînd, rînd, lin, col, tip)
#define D3FREE( prînd)
```

Se ia ca model metoda prezentată pentru șirul bidimensional.

15. Să se modifice funcția *idim3()* pentru a crea un șir tridimensional de orice tip. Următorul segment de cod ilustrează apelul funcției *idim3()* pentru crearea a două șiruri tridimensionale, unul de întregi de dimensiune *3x5x6*, iar altul de *double* de *4x6x8*.

```
...
char ***idim3();
int ***a;
double ***b;

a = (int ***) idim3( 3, 5, 6, sizeof(int));
b = (double ***) idim3( 4, 6, 8, sizeof(double));
...
```

16. Să se completeze macroul pentru accesul la un element al unui șir tridimensional:

```
#define DIMENSION3( arga, usera, rînd, lin, col, tip)...
```

unde *arga* este pointerul la șirul tridimensional existent, ce conține *rînd* rînduri, *lin* linii și *col* coloane. Macroul creează un nou șir tridimensional de tipul *tip*. Se ia ca model metoda prezentată la șiruri bidimensionale.

4. O privire generală asupra limbajului C

Limbajul C ne permite descrierea unor operații sau funcții care sînt specifice limbajelor de asamblare. Astfel se pot scrie programe care atribuie variabilelor valorile regiștrilor calculatorului, memorează valorile variabilelor în regiștrii, manevrează adrese de memorie sau schimbă dinamic tipurile de date. Toate aceste operații generează un limbaj puternic, C-ul cerînd o corectitudine aparte în aplicare. Lipsa erorilor de compilare nu indică neapărat un program corect.

Capitolul de față prezintă modul în care compilatorul evaluează expresii, o tehnică de descifrare a declarațiilor complexe cu șiruri, funcții și pointeri, precum și modul de utilizare a șirurilor în expresii de atribuire sau ca argumente în funcții.

4.1. Secvențe garantate

Expresiile C care conțin operatorii ++ și -- se vor evalua diferit pe diverse configurații de calculatoare. Din acest motiv aceste expresii nu sînt portabile.

Multe expresii C conțin operatori ce interconectează subexpresii. Dacă o subexpresie modifică o variabilă ce este utilizată în altă subexpresie, evaluarea întregii expresii nu este întotdeauna cea așteptată (previzibilă). Iată cîteva exemple edificatoare în acest sens. Instrucțiunea de atribuire:

```
a[i++] = i;
```

s-ar putea să nu memoreze valoarea dorită a lui *i* în șirul *a*, iar expresia :

```
*p-- + *p
```

s-ar putea să folosească greșit pointerul în încercarea de a aduna valorile de la două adrese.

Pe de altă parte evaluarea expresiei:

```
*p++ == '\t' ? ' ' : *p
```

este unic determinată. Rezultatul instrucțiunii de atribuire:

```
cost[i++].membru += valoare;
```

este neambiguu, în timp ce la 'echivalenta' ei:

```
cost[i++].membru = cost[i++].membru + valoare;
```

rezultatul este ambiguu.

Este însă posibil ca evaluarea acestor expresii să producă rezultatul scontat, caz în care funcționarea defectuoasă nu este observată.

4.1.1. Operatorii secvențelor garantate

Expresiile C pot conține subexpresii separate prin operatori. Dacă aceeași variabilă apare în mai mult de o subexpresie, ordinea de evaluare devine importantă în cazul în care operatorul modifică variabilele. De exemplu, expresia: *a[i++] + b[i]* conține două subexpresii care ambele conțin referințe la șiruri. Prima subexpresie conține operatorul ++ și modifică valoarea lui *i*. Evaluarea expresiei nu este unică, deoarece ea depinde de momentul în care se incrementează variabila *i*. Compilatorul poate aduna pe *a[i]* cu *b[i]* sau pe *a[i]* cu *b[i+1]*.

Tabelul 4.1 prezintă operatorii C care modifică variabilele și produc așa numitul efect lateral (sau secundar).

Operatori	Semnificatie
++ si -- = += -= *= /= %= <<= >>= &= |= ^=	pre/post increment/decrement atribuire atribuire aritmetica atribuire pe bit

Tabelul 4.1 Operatori cu efect lateral

Dacă se utilizează într-o subexpresie un operator cu efect lateral trebuie să se cunoască ordinea de evaluare a operatorilor.

Există un set mic de operatori care interconectează subexpresii cu operatori cu efect lateral și garantează că întreaga expresie se va evalua identic pe orice calculator. Numim această expresie **secvență garantată** de operatori. Există doar patru operatori de acest tip:

Operatori	Semnificatie
&& || , ? :	SI logic SAU logic , (virgula) conditional

Tabelul 4.2. Operatorii secvențelor garantate

Secvențele garantate de operatori sînt importante, deoarece ele evaluează subexpresiile de la stînga la dreapta. Ordinea evaluării subexpresiilor fiind bine determinată, subexpresiile pot utiliza operatori cu efect lateral. În timpul evaluării unei subexpresii cu un operator cu efect lateral, trebuie să se modifice variabila înaintea întîlnirii unui operator de secvență garantată.
Pentru a evidenția acest lucru, vom analiza operatorii secvenței garantate pe exemple.

4.1.2. Operatori logici (&& și ||)

Considerăm formatul pentru ȘI logic de forma:

```
expresie_1 && expresie_2 && ...
```

Pentru ca expresia totală să fie evaluată la valoarea de adevăr *TRUE*, trebuie ca fiecare subexpresie să fie evaluată la valoarea de adevăr *TRUE*. Evaluarea începe de la stînga la dreapta și, dacă o subexpresie este evaluată la *FALSE*, nici o subexpresie din dreapta nu se mai evaluează.

Acest lucru poate fi utilizat avantajos în exemplul:

```
d! == 0.0 && printf(" %g \ n", b/d);
```

care evită eventuala împărțire la zero; sau în exemplul:

```
debug && dump(buf);
```

ce apelează funcția *dump()* doar dacă fanionul de depanare (*debug*) are valoarea *TRUE*.

Acest lucru este util în apelul funcțiilor ce returnează *TRUE* (diferit de zero) sau *FALSE* (zero). De exemplu, instrucțiunea:

```
if (verif1(buf) && verif2(buf) && verif3(buf))
        prelucr(buf);
```

va prelucra șirul *buf*, dacă acest șir a fost verificat în ordine de funcțiile `verif1()`, `verif2()` și `verif3()` și fiecare dintre ele a returnat *TRUE*. Ca atare, este posibilă înlocuirea instrucțiunii cu:

```
verif1(buf) && verif2(buf) && verif3(buf) && prelucr(buf);
```

(se poate scrie așa atîta timp cît funcția *prelucr()* nu e de tip *void*).
Să considerăm macrourile ce folosesc operatorul logic *&&* pentru lucrul cu șiruri:

```
#define STRCMP(a,b) ( *(a)==*(b) && strcmp(a,b)==0)
#define STRNCMP(a,b,n) ( *(a)==*(b) && strncmp(a,b,n)==0)
```

Fiecare macrou compară primul caracter din fiecare șir și apelează funcția de bibliotecă corespunzătoare doar dacă ele sînt egale. Macroul elimină apelul funcției dacă primul caracter din fiecare șir este diferit.

Vom descrie modul în care se tratează operatorii cu efect lateral ++ și -- din subexpresii interconectate. Presupunem că se parcurg elementele unui șir, pînă cînd se găsește un element mai mare decît o valoare dată. Elementul următor se transmite ca argument unei funcții. Expresia este:

```
a[i++] > VAL && calc( a[i]);
```

Acum este sigur că incrementarea variabilei *i* se face exact după testul de comparare cu *VAL*.

În ceea ce privește operatorul SAU (||) din expresia:

```
expresie_1 || expresie_2 || ...
```

este suficientă o singură evaluare a unei expresii la *TRUE*, pentru ca toată expresia să fie adevărată. Acest lucru se poate exploata în exemple de felul :

```
p == NULL || concat(p, listă)
```


Acest exemplu previne concatenarea elementului referit de pointerul *p* la o listă, dacă pointerul este vid (element inexistent). Prin aceste construcții se cîștigă și viteză în execuție.

4.1.3. Operatorul virgulă (,)

C-ul permite separarea prin virgulă a oricîte subexpresii de forma:

```
expresie_1 , expresie_2 , ...
```

Ordinea de evaluare este de la stînga la dreapta. Subexpresiile pot conține operatori cu efect lateral.

De exemplu, programul **rev1.c** inversează caracterele dintr-un șir utilizînd operatorul virgulă în interiorul unei bucle *for*.

Programul 4.1

```
/* rev1.c - inversează un șir de caractere */

#include <stdio.h>

main()
{
        char *s="eram";
        reverse( s);
        printf("%s\n", s);
}

reverse( register char *p)
{
        register char *q=p+strlen( p)-1;
        for ( ;p<q; p++, q--) {
                *p ^= *q;
                *q ^= *p;
                *p ^= *q;
        }
}
```

Am utilizat operatorul virgulă pentru a incrementa pointerul *p* și a decrementa pointerul *q* după fiecare execuție a buclei.

O altă utilizare a operatorului virgulă este în subexpresii formate dintr-o singură instrucțiune. Considerăm bucla din programul care interschimbă elementele adiacente ale unui șir, funcție de valoarea fanionului *flag*.

```
if (flag)
        tmp = x[i++], x[i-1] = x[i], x[i]=tmp;
```

Dacă fanionul *flag* este evaluat la *TRUE*, variabila *i* se utilizează pentru a accesa elementele dorite din șirul *x*. Variabila *tmp* e necesară pentru a interschimba elementele adiacente. Se observă că variabila *i* se incrementează după evaluarea primei subexpresii. Am evitat astfel blocul necesar instrucțiunii *if*.

4.1.4. Operatorul condițional (? :)

Acest operator conține trei subexpresii:

```
expresie_1 ? expresie_2 : expresie_3
```

Subexpresiile se evaluează de la stînga la dreapta. Dacă *expresie_1* este adevărată (nonzero) se evaluează *expresie_2*, altfel *expresie_3*. Efectul lateral poate să apară în orice subexpresie.

Operatorul condițional este utilizat în instrucțiuni *return*, în macrouri și în lista argumentelor de funcții. De exemplu, funcția *printf()* de mai jos afișează un număr în format hexazecimal sau zecimal, funcție de fanionul *mod*:

```
printf(mod == HEX ? "%x\n" : "%d\n", num);
```

Expresie_2 și *expresia_3* sînt în acest caz constante șir.
Declarația:

```
unsigned char buf[sizeof(int) == 4 ? 1024 : 512];
```

definește șirul *buf* de *1024* octeți pentru calculatoare ce reprezintă tipul întreg pe *32* de biți, și de *512* octeți în alte cazuri.

Compilatorul utilizează operatorul condițional ca secvență garantată în expresii ce conțin subexpresii cu operatori cu efect lateral. Acest lucru putem să îl demonstrăm prin exemplul:

```
int *p;
...
timp(*p = ++ *p < 60 ? *p : 0);
```


Se consideră că prin intermediul pointerului *p* se numără secundele. Conținutul pointerului se incrementează la fiecare secundă și se apelează funcția *timp()* pentru afișarea timpului. Limbajul C garantează ordinea corectă de evaluare, deoarece operatorul condițional are precedență mai mare decît cel de atribuire. Ca atare, dacă conținutul pointerului este mai mic decît *60*, funcția *timp()* afișează noua valoare, în caz contrar conținutul este inițializat cu valoarea *0*.

O confuzie poate să apară între operatorul virgulă și separatorul virgulă. Operatorul virgulă garantează evaluarea de la stînga la dreapta, iar separatorul virgulă nu. De exemplu, apelul:

```
f(rînd = *p++ , col = *p++);
```

nu este portabil, deoarece limbajul C nu garantează că parametrii funcției se evaluează de la stînga la dreapta. Aceste probleme nu sînt întotdeauna evidente. Următorul segment de program poate să se execute corect pe un calculator și incorect pe un altul:

```
int a[MAX];
int i = 0;
...
while ( i < MAX )
        printf("%4d%c", a[i], ++i % 10 ? ' ': '\n');
```

Programul afișează cîte *10* elemente ale unui șir pe o linie. Problema apare cu *a[i]* și cu operatorul ++ aplicat variabilei *i*. Compilatorul poate incrementa variabila *i* înainte de referirea unui element din șir. Programul nu este portabil, deoarece argumentele funcției se pot evalua în orice ordine.

O soluție portabilă este:

```
int a[MAX];
int i;
...
for (i = 0; i < MAX; i++) {
        printf("%4d%c", a[i], (i+1) % 10 ? ' ': '\n');
```

unde s-a eliminat operatorul cu efect lateral din lista argumentelor funcției *printf()*.

4.2. Regula dreapta-stînga

Limbajul C foloseşte declaraţii ce pot include pointeri, funcţii sau şiruri. Sintaxa unor astfel de declaraţii pare confuză la prima vedere.

Regula dreapta-stînga este cea care explică modul în care compilatorul tratează o declaraţie complexă. Folosindu-ne de ea, putem, la rîndul nostru, să construim propriile declaraţii şi să le descifrăm pe cele existente.

4.2.1. Declaraţii complexe

O declaraţie ce include orice combinaţie de pointeri, şiruri şi funcţii o considerăm o declaraţie complexă. Pentru a analiza o declaraţie complexă, compilatorul trebuie să separe atributele declaraţiei. Există trei atribute:

Atribut	Semnificatie
() [] *	functie sir pointer

Tabelul 4.3. Atributele unei declaraţii

Precedenţa atributelor scade în tabel, iar pentru atributele *()* şi *[]*, care au aceeaşi precedenţă, se aplică evaluarea de la stînga la dreapta.
O listă de declaraţii corecte ar fi:

Combinatie de atribute	Semnificatie
*() (*)() *[] (*)[] [][] **	functie ce returneaza pointer pointer la functie sir de pointeri pointer la sir tablou bidimensional (matrice pointer la pointer

Tabelul 4.4. Declaraţii corecte

Lista de declaraţii se poate extinde, mărind complexitatea. De exemplu, **[][]* este un masiv bidimensional de pointeri.
Operatorul de indirectare * este pus între paranteze în declaraţia unui pointer la o funcţie sau la un şir datorită precedenţei sale. Nu toate combinaţiile de atribute sînt declaraţii corecte:

Atribute de atribute	Semnificatie
()[] ()[] ()()	functie ce returneaza un sir sir de functii functie ce returneaza o functie

Tabelul 4.5. Declaraţii incorecte

Aceste declaraţii devin corecte dacă se returnează pointer la un şir sau pointer la o funcţie.

Prima observaţie ce se impune într-o declaraţie complexă este:

Separarea atributelor *()* şi *[]* prin utilizarea parantezelor '(' şi ')'.

Această specificare rezultă din faptul că C-ul nu permite declararea unui şir de funcţii sau a unei funcţii ce returnează şir. Dacă nu se ţine cont de această observaţie, compilatorul semnalează eroare la compilare.

A doua observaţie foarte utilă la descifrarea unei declaraţii complexe este:

Substituirea atributelor prin anumite şabloane text:

Atribut	Şablon text
()	funcţia returnează
[n]	şir de n
*	pointer la

Tabelul 4.6 Şabloane text ce echivalează un atribut

4.2.1. Aplicarea regulii dreapta-stînga

La descifrarea (decodificarea) unei declarații complexe se respectă pașii:

1) Se începe cu identificatorul.
2) Se caută în dreapta identificatorului un atribut.
3) Dacă nu există nimic, se caută în partea stîngă.
4) Dacă există, se substituie atributul cu șablonul corespunzător.
5) Se continuă substituția dreapta-stînga.
6) Descifrarea declarației se oprește la întîlnirea tipului datei.

Regula este aplicabilă atît declarațiilor simple, cît și celor complexe.

Exemple:

```
unsigned long buf[10];
```

- *buf* este identificatorul; în dreapta există atributul *[]*. Deci *buf* este un șir de zece întregi de tip *long*, fără semn.

```
unsigned int *p[2];
```

- *p* este identificatorul; în dreapta există atributul *[]*, deci *p* este un șir de doi pointeri; în stînga există atributul * și, ca atare, *p* este un șir de doi pointeri la întregi fără semn.

```
struct orice *ps[3][4];
```

- *ps* este identificator; în dreapta există atributul *[]* și din nou atributul *[]*; în stînga există atributul *. Ca atare, *ps* e un șir bidimensional de pointeri la structura *orice*. Altfel spus, *ps* e un masiv (tablou) de *3x4* pointeri la structura *orice*.

```
int (*pf[5])();     /* declarație complexă */
```

- *pf* e identificatorul; în dreapta există atributul *[]* deci, *pf* este un șir. În stînga există atributul *, deci *pf* este un șir de pointeri. La dreapta atributul *()* precizează că fiecare element este o adresă la o funcție ce returnează un întreg. Deci *pf* este un șir de cinci pointeri la funcții ce returnează întregi.

```
int *(*pf[5])();   /* declarație complexă */
```


Modificarea față de exemplul precedent apare doar la tipul returnat, care în acest caz devine pointer la întreg.

```
struct bloc *(*tablou())[10];
```

- *tablou* este o funcție ce returnează un pointer la un șir de zece pointeri la structuri de tipul *bloc* (se aplică prima observație).

Cît privește operația inversă, de construcție a unei declarații complexe, se pornește cu analiza cerinței și se aplică regulile enunțate. De exemplu:

a) dorim un șir de zece pointeri la funcții ce returnează pointeri la întregi. Secvența de obținere a declarației finale respectă pașii:

```
buf[10]; /* șir de zece elemente */
*buf[10];          /* șir de zece pointeri */
```

Urmează 'la funcții' și ca atare se pun parantezele și atributul *()*, conform primei observații:

```
(*buf[10])()       /* șir de pointeri la funcții */
int *(*buf[10])() /* șir de zece pointeri la funcții ce */
                   /* returnează pointer la întreg */
```

b) dorim un pointer la o funcție ce returnează un pointer la un masiv bidimensional de trei linii și patru coloane. Secvența de obținere a declarației este:

```
*pf;               /* pf pointer */
(*pf)(); /* pf pointer la funcție */
*(*pf)() /* pf pointer la funcție ce returnează un */
                   /* pointer */
double (*(*pf)())[3][4];
```

4.2.2. Operații pe șiruri

Se combină elementele prezentate anterior într-un exemplu ce permite implementarea unui limbaj cu operatori pe șiruri. Presupunem că limbajul acceptă doi operatori + și -. Ambii operatori realizează concatenarea a două

şiruri. Deosebirea dintre ei constă în faptul că al doilea inserează un blanc între şirurile concatenate.

Implementarea în C a acestui exemplu permite utilizarea unui şir de pointeri la funcţii. Operatorul trebuie convertit într-un index în şirul pointerilor la funcţii şi apoi se apelează funcţia corespunzătoare operatorului. Programul următor prezintă modul de implementare a operatorilor + şi -.

Program 4.2.

```
/* şir1.c - concatenarea şirurilor cu pointeri la funcţii */
#include <stdio.h>
char *malloc(), *strcpy(), *strcat();
main()
{
        char *join(), *joinb();
        char *(*ps[2])();
        ps[0]=join;
        ps[1]=joinb;
        printf("%s\n", (*ps[0])("alfa", "bet"));
        printf("%s\n", (*ps[1])("alfa", "beta"));
}
char *join( char *s1, char *s2)
{
        char *p;
        p = (char *)malloc( strlen( s1) + strlen( s2) +1);
        if ( p== (char *) NULL) {
                fprintf( stderr, "Eroare malloc\n");
                exit(1);
        }
        return strcat( strcpy( p, s1), s2);
}
char *joinb( char *s1, char *s2)
{
        char *p;
        p = (char *)malloc( strlen( s1) + strlen( s2) +2);
        if ( p== (char *) NULL) {
                fprintf( stderr, "Eroare malloc\n");
                exit(1);
        }
        return strcat( strcat( strcpy( p, s1)," "), s2);
}
```


Execuţia programului afişează:

```
alfabet
alfa beta
```

Declaraţia:

```
char *(*ps[2])();
```

creează un şir de doi pointeri la funcţii ce returnează pointer la tipul *char*. Instrucţiunile:

```
ps[0]=join;
ps[1]=joinb;
```

iniţializează cele două elemente ale şirului cu adresele funcţiilor. Apelul funcţiilor este realizat prin instrucţiunile:

```
(*ps[0])("alfa", "bet")              /* apel join */
(*ps[1])("alfa", "beta")    /* apel joinb */
```

Ambele funcţii apelează cîteva funcţii de bibliotecă. Funcţia *malloc()* alocă memoria necesară şirului rezultat din concatenarea celor două şiruri argument din apel. Funcţia *strcpy()* copiază şirul ce apare ca al doilea argument în şirul ce apare ca prim argument, iar funcţia *strcat()* concatenează şirurile argument. Funcţiile *strcpy()* şi *strcat()* returnează un pointer la zona heap unde există memorat şirul rezultat. Programul principal foloseşte aceşti pointeri pentru a afişa prin funcţia *printf()* şirul concatenat.

4.2.3. Semnale

Programele C pot comunica între ele prin semnale. Un program poate transmite sau recepţiona semnale de la el însuşi sau de la alte programe. Modul în care programul tratează semnalul este dependent de aplicaţie, putîndu-se opta pentru a ignora semnalele sau pentru a executa funcţii proprii dependente de semnalele recepţionate.

În contextul celor anterior discutate prezintă interes declaraţia funcţiei *signal()*. În manualul de utilizare TC ea apare sub forma:

```
#include <signal.h>
void (* signal( int sig, void (* func)( int sig[, int subcod])))( int);
```

Această declarație este suficient de complexă pentru a determina pe unii programatori să renunțe la folosirea semnalelor. Aplicînd regula dreapta-stînga se deduce că *signal* este o funcție de doi parametri, care returnează o valoare fără tip (*void*). Primul parametru, *sig*, este un întreg ce reprezintă numărul unui semnal, iar al doilea, *func*, prin aplicarea aceleași reguli, este un pointer la o funcție ce returnează tot o valoare fără tip.

Asupra lucrului cu semnale se va reveni în capitolul 5.

4.2.4. Castul

Tabelul 4.7 ilustrează cîteva declarații complexe cu funcții, șiruri și pointeri. Restricțiile impuse la declarațiile complexe sînt următoarele:

- funcțiile nu pot returna șiruri, structuri, uniuni sau funcții, dar ele pot returna pointeri la astfel de obiecte,
- nu există șir de funcții, dar există șir de pointeri la funcții,
- nu există structură sau uniune ce conține o funcție, dar ea poate să conțină un pointer la funcție.

Atribut	Semnificatie
(*())[]	functie ce returneaza un pointer la sir
(*[])()	sir de pointeri la functii
(*())()	functie ce returneaza pointer la functie

Tabelul 4.7. Declarații complexe corecte

Pentru descifrarea unei astfel de declarații, se aplică regula dreapta-stînga. De exemplu, în declarația:

(*())[]

deoarece identificatorul lipsește, atributul *()* indică o funcție. Atributul * din partea stîngă indică un pointer, iar atributul *[]* din afara parantezei conduce în final la o funcție ce returnează un pointer la un șir.

Modelul de interpretare a acestor declarații prin regula dreapta-stînga este aplicabil și castului. De exemplu, apelul:

signal(SIGQUIT, SIGIGN);

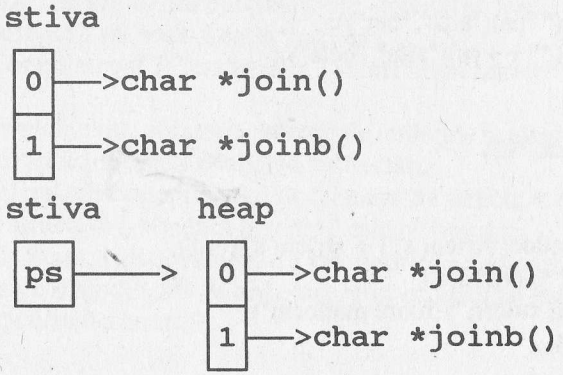
determină ignorarea semnalului *SIGQUIT*. Întrucît funcția *signal()* necesită pe poziția argumentului al doilea un pointer, definiția constantei simbolice *SIGIGN* trebuie să fie corespunzătoare. Fișierul **signal.h** conține declarația:

#define SIGIGN (void (*)()) 1

ce aplică castul numărului *1*. Regula dreapta-stînga aplicată castului demonstrează echivalența acestuia cu definiția argumentului al doilea din definiția funcției *signal()*.

Alocarea dinamică este un alt caz de utilizare a castului pentru conversia unui tip de dată. De exemplu, foarte des este necesară conversia tipului pointerului returnat de funcția *malloc()*.

Programul **str1.c** conține un șir de pointeri la funcții care returnează pointer la caracter. În acest caz șirul *ps* există în zona stivă. Dacă alocarea trebuie să fie dinamică este necesar un pointer la șir. Schematic cele două situații ar arăta astfel:



Pointerul *ps* este în zona stivă și conține o adresă heap. Ca atare, *ps* este un pointer la un pointer la o funcție ce returnează un pointer. Regula dreapta-stîngă conduce la declarația:

char * (** ps)();

Din acest motiv la apelul funcției *malloc()* trebuie utilizat un cast analog la atribuirea pointerului *ps*. Programul modificat este ilustrat mai jos:

Programul 4.3.

```
/* șir2.c - concatenarea șirurilor cu pointeri la funcții din zona heap
*/

#include <stdio.h>
char *malloc(), *strcpy(), *strcat();

main()
{
        char *join(), *joinb();
        char *(**ps)();
        ps = (char *(**)()) malloc( 2*sizeof( char *));
        if ( ps == (char *(**)()) NULL) {
                fprintf( stderr, "Eroare malloc ps\n");
                exit(1);
        }
        ps[0]=join;
        ps[1]=joinb;
        printf("%s\n", (**ps)("alfa", "bet"));
        printf("%s\n", (**++ps)("alfa", "beta"));
}

char *join( char *s1, char *s2)
{
        char *p;
        p = (char *)malloc( strlen( s1) + strlen( s2) +1);
        if ( p== (char *) NULL) {
                fprintf( stderr, "Eroare malloc\n");
                exit(1);
        }
        return strcat( strcpy( p, s1), s2);
}

char *joinb( char *s1, char *s2)
{
        char *p;
        p = (char *)malloc( strlen( s1) + strlen( s2) +2);
```



```
        if ( p== (char *) NULL) {
                fprintf( stderr, "Eroare malloc\n");
                exit(1);
        }
        return strcat( strcat( strcpy( p, s1)," "), s2);
}
```

Castul care se aplică valorii returnate de funcția *malloc()* modifică pointerul la *void* în pointer la pointer la funcție, ce returnează un pointer la *char*. Castul este identic cu declarația lui *ps*, dar lipsește identificatorul.

4.3. L-valori în expresii

Termenul de *l-valoare* apare frecvent în documentația de specialitate și în mesajele de eroare afișate la compilare.

O l-valoare (valoare stînga) este o entitate ce apare în partea stîngă a unei instrucțiuni de atribuire. Bineînțeles, noțiunea e mult mai largă, dar limbajul C face o mare distincție între o l-valoare și o r-valoare (adică o entitate ce apare în partea dreaptă a unei instrucțiuni de atribuire). L-valoarea are un sens aparte, deoarece ei îi este asociată o adresă de memorie. Altfel spus, ea poate memora o valoare. De exemplu, instrucțiunea de atribuire: 9=7; nu este compilabilă, întrucît întregul *9* nu are o locație asociată.

Compilatorul TC tratează ca l-valori următoarele:

- numele variabilelor, pointerii, structurile, uniunile și elementele șirurilor;
- rezultatul aplicării operatorului de indirectare;
- rezultatul aplicării operatorului de membru de structură (.) sau operatorului de pointer la structură (->), dacă membrul este o l-valoare.

Ca atare, există doar patru operatori de l-valoare: *[]*, *, ., ->. Orice l-valoare poate fi modificată de orice operator cu efect lateral (vezi tabelul 4.1). De exemplu, pentru declarațiile:

```
int num;
char c, *p;
float val[10];
struct s {
        char buf[10];
        float *pd;
} date;
struct s *ps = &date;
```

expresiile de mai jos folosesc corect l-valorea:

```
num += 5;
p = "şir nou";
*p = 'x';
val[4] = 6.5;
date.pd = &val[4];
ps->buf[1] = *p;
```

Confuzia apare la ceea ce nu este considerată o l-valoare. Compilatorul consideră r-valori:

- constantele
- numele funcţiilor
- valorile enumerate
- numele de şiruri

R-valorile pot fi referite, dar nu modificate. Este evidentă imposibilitatea utilizării unui nume de funcţie ca l-valoare, deoarece ea nu are o adresă de memorie pe care să o putem modifica. Adresele de funcţii din zona text sînt de regulă protejate la operaţii de scriere. Analog, valorile enumerate fiind constante, programele nu le pot modifica din aceleaşi considerente. Numele şirurilor creează confuzie, deoarece limbajul permite interschimbarea şirurilor cu pointeri. Pentru a clarifica acest lucru, considerăm programul:

Programul 4.4.

```
#include <stdio.h>
void f(), g();

main()
{
        static char a[] = "acesta este un şir";
        char *p = "acesta este un ALT şir";
        /* instrucţiunea următoare nu se compilează
        a = "acesta este un şir NOU"; */
        p = "acesta este un ALT şir NOU";
        printf("pointerul p este: %x la ->%s\n \
                adresa şirului referit de p: %x\n", &p, p, p);
        f( a);
        printf("şirul a este: %s\n", a);
```



```
        g( &p);
        printf("pointerul p este: %x la ->%s\n \
                adresa şirului referit de p: %x\n", &p, p, p);
}

void f( char a[])
{
        a = "nu pot schimba şirul";
        printf("şirul a în f() este: %s\n", a);
}

void g( char **ptr)
{
        *ptr = "am reuşit să schimb şirul";
}
```

Execuţia programului afişează:

```
pointerul p este: ffda la ->acesta este un ALT şir NOU
                adresa şirului referit de p: 1be
şirul a în f() este: nu pot schimba şirul
şirul a este: acesta este un şir
pointerul p este: ffda la ->am reusit sa schimb şirul
                adresa şirului referit de p: 2a0
```

Acest rezultat pune puţin pe gînduri chiar programatorii experimentaţi. După iniţializarea şirului *a* se iniţializează pointerul *p* cu adresa constantei şir *"acesta este un ALT şir"*. În timpul execuţiei `áirul` *a* se găseşte în zona de date, iar *p* în zona stivă.

Pointerul *p* are atribuită o locaţie de memorie pe cînd şirul *a* nu are, deoarece numele şirului este o constantă. Acest lucru nu este în contradicţie cu transmiterea adresei unui şir ca argument pentru o funcţie (de exemplu *f(a)*), deoarece funcţiei se transmite adresa primului element din şir.

În interiorul funcţiei *f()* avem o instrucţiune similară celei din funcţia *main()*. Instrucţiunea:

```
a = "nu pot schimba şirul";
```

este compilată, spre deosebire de cea din *main()*, care nu este. Explicaţia este simplă. Dacă în *main()* numele şirului *a* era privit ca şi constantă, în funcţia *f()* se foloseşte un alt pointer numit *a*. La apelul funcţiei *f()* se alocă o locaţie separată

pentru un pointer ce conține adresa de început a șirului. Ca atare, în interiorul funcției *f()* se poate utiliza notația *a[]*, dar e ca și cum s-ar utiliza un pointer. Cum *a* este l-valoare în interiorul funcției *f()* ea nu se va putea modifica. Instrucțiunea *printf()* ce urmează apelului funcției *f()* demonstrează schimbarea copiei locale a pointerului. La revenirea în programul principal se constată că nu s-a modificat conținutul șirului inițial. Pentru a-l modifica se apelează funcția *g()* cu adresa pointerului la șir. Deoarece *ptr* este declarat pointer la pointer se modifică pointerul din programul principal astfel încît el va referi o altă constantă șir (ce are altă adresă).

4.3.1. L-valori în expresii compacte cu pointeri

Cele mai utilizate expresii compacte cu pointeri sînt ilustrate mai jos:

Incrementare	Decrementare	L-valoare
*p++	*p--	da
*++p	*--p	da
++*p	--*p	nu
(*p)++	(*p)--	nu

Tabelul 4.8 L-valoare și expresii compacte cu pointeri

Din cîte se observă, doar primele două linii din tabel sînt l-valori. Pentru a demonstra aceasta, fie declarațiile:

```
int buf[20], *p=buf;
```

În instrucțiunile:

```
*p++ = 7;/*inițializează primul element al șirului buf cu 7 */
*++p = 9;/*inițializează al treilea element al șirului cu 9 */
```

expresiile compacte cu pointeri sînt l-valori. Nu același lucru se poate spune despre instrucțiunile:

```
++*p = 7;
(*p)++ = 9;
```


Acestea sînt incorecte și vor genera erori la compilare, deoarece operatorul ++ se aplică obiectului referit.

4.4. Transmiterea șirurilor

În capitolul 3 s-a arătat că limbajul C convertește referința la șir în pointer cu deplasament. Numele unui pointer este l-valoare, dar numele șirului nu.

Fie *a* și *b* două șiruri de același tip și dimensiune. Atribuirea:

```
a = b;
```

produce eroare de compilare, deoarece șirul *a* nu este o l-valoare. Transferul datelor între șiruri (atribuirea șirurilor) nu este permisă în limbajul C.

În schimb, limbajul C permite atribuirea unei structuri la alta. Dacă *s* și *t* sînt două structuri de același tip, atribuirea:

```
s = t;
```

este compilată, deoarece *s* este l-valoare. Ca atare, este posibil transferul tuturor membrilor dintr-o structură sau uniune în alta (atribuirea structurilor și uniunilor).

La atribuirea structurilor compilatorul generează următorul cod în limbaj de asamblare:

```
lea di, &s
lea si, &t
mov cx, sizeof(s)/2
rep movsw
```

În acest caz atribuirea structurilor se execută rapid, deoarece nu există bucle. Dacă însă codul generat în limbaj de asamblare nu conține instrucțiuni de transfer de bloc (*rep movsw*), ci bucle, atribuirea se va executa mai lent. Rezultă că atribuirea structurilor este dependentă de calculator.

Folosind atribuirea structurilor se poate realiza și atribuirea șirurilor. Condiția este ca șirurile sursă și destinație să fie membrii unei structuri de același tip.

4.4.1. Tehnica

Presupunem două şiruri de acelaşi tip şi aceeaşi lungime:

```
tip c[DIM], d[DIM];
```

unde *DIM* este o constantă simbolică, definită printr-o directivă *define*. Pentru a realiza atribuirea şirurilor se declară structura şablon:

```
struct şabl {
        tip buf[DIM];
};
```

Pentru aceasta structură compilatorul nu alocă memorie. Fie declaraţia:

```
struct şabl a, b;
```

Instrucţiunea:

```
b = a;
```

transferă datele din structura *a* în structura *b*.
Atribuirea structurilor poate fi realizată şi prin intermediul pointerilor la structuri. Pentru declaraţiile:

```
struct şabl *pa = &a;
struct şabl *pb = &b;
```

transferul datelor din structura *a* în structura *b* se va realiza prin atribuirea:

```
*pa = *pb;
```

Combinînd cele două operaţii într-o singură instrucţiune se obţine:

```
*( struct şabl *)d = *( struct şabl *) c;
```

Această atribuire se realizează practic în doi paşi. La primul pas *d* devine pointer la structura şablon *şabl*, iar la al doilea, prin aplicarea operatorului de indirectare, se realizează atribuirea de structuri prin pointerii la structuri.

Cu toate că tehnica este foarte eficientă ea nu este recomandată, deoarece nu garantează că acest transfer de date este portabil pentru toate implementările C. Recomandăm însă utilizarea ei în Turbo C.

Dacă portabilitatea este prioritară în programul dumneavoastră, se pot utiliza în locul tehnicii prezentate funcţii de bibliotecă similare funcţiei *memcpy()*.

Prezentăm tehnica transferului rapid în următorul program, care transferă cinci întregi din şirul *a* în şirul *b*.

Programul 4.6.

```
#include <stdio.h>
#define MAX 5

main()
{
        struct num {
                int buf[MAX];
        };
        static int a[MAX]= { 1, 2, 3, 4, 5};
        static int b[MAX]= { 5, 4, 3, 2, 1};

        int i;
        *(struct num *) b = *(struct num *) a;

        for ( i=0; i<MAX; i++)
                printf("%2d", b[i]);
        putchar('\n');
}
```

Iniţial s-a declarat o structură şablon *num*, care conţine şirul *buf*. Şirul *a* conţine cinci numere în ordine crescătoare, iar şirul *b* aceleaşi numere, dar în ordine descrescătoare. Instrucţiunea:

```
*( struct num *) b = *( struct num *) a;
```

transferă cei cinci întregi din şirul *a* în şirul *b*. În final programul afişează conţinutul şirului *b*.

4.4.2. O funcție care transferă blocuri

Se aplică tehnica precedentă la implementarea unei funcții, care transferă blocuri de date în memorie (similară funcției *memcpy()*). Fie două șiruri de caractere declarate prin:

```
unsigned char a[DIM], b[DIM];
```

Posibile apeluri ale funcției `bmove ( )` sînt:

```
unsigned nr, i;
...
/* mută tot șirul a în șirul b */
bmove( b, a, sizeof(a));
...
/* mută 100 de elemente din șirul b în șirul a */
bmove( a, b, 100*sizeof( a[0]));
...
/* mută nr elemente din a[i] și b[i] */
bmove( &b[i], &a[i], nr*sizeof( a[i]));
```

Primele două argumente ale funcției *bmove()* sînt pointeri la șirul sursă respectiv destinație, iar ultimul este numărul de elemente ce se transferă.

Programul 4.7

```
/* bmove.c */

#define DIM 5
#define BUFSIZE 256
bmove( unsigned char *d, unsigned char *s, int cont)
{
        struct bloc {
                char a[BUFSIZE];
        };
        while ( cont >= BUFSIZE) {
                *( struct bloc *) d = *( struct bloc *) s;
                d += BUFSIZE;
                s += BUFSIZE;
                cont -= BUFSIZE;
        }
```



```
        while ( cont --)
                *d++ = *s++;
}

#include <stdio.h>
main()
{
        int i;
        unsigned char a[DIM], b[DIM]={ 1, 2, 3, 4, 5};
        bmove( a, b, sizeof( a));
        for (i=0;i<DIM;i++)
                printf(" %d ", a[i]);
        putchar('\n');
}
```

Prima buclă compară variabila contor *cont* cu constanta *BUFSIZE*, efectuînd un transfer pe blocuri atîta timp cît *cont* nu devine mai mic decît lungimea unui bloc transferabil. La un moment se transferă un bloc de *256* octeți, după care se incrementează atît pointerul la șirul sursă cît și cel la șirul destinație și se decrementează contorul cu valoarea *BUFSIZE*.

A doua buclă *while* termină transferul octeților rămași din ultimul bloc incomplet ocupat. Motivul utilizării blocurilor de pînă la *256* de octeți are ca scop minimizarea timpului de execuție din buclă.

Funcția *bmove()* transferă datele în memorie mai eficient decît funcția de bibliotecă *memcpy()*. Cu toate acestea funcția *bmove()* rămîne doar o simplă aplicație la transferul rapid de șiruri și nu are scopul de a înlocui funcția *memcpy()* în programele C portabile.

4.5. Transmiterea șirurilor ca parametri pentru funcții

Limbajul C transmite argumentele la funcții prin valoare (call by value). La apelul unei funcții trebuie să fie clar ce reprezintă acea valoare: valoarea unei variabile sau adresa ei în memorie. Este normal ca atunci cînd transmitem valoarea unei variabile la o funcție ce așteaptă o adresă (sau invers) programul să nu se execute corect.

De regulă, dacă o funcție așteaptă o adresă ca argument, apelul funcției se face cu operatorul *&* în fața argumentului respectiv. Funcția declară un pointer pentru a accesa data și poate modifica data originală cu acest pointer. Dacă, însă, funcția așteaptă o valoare ca argument, apelul funcției se face fără operatorul *&* în fața argumentului, iar funcția declară o variabilă locală ce conține o copie a

datei, funcţia neputînd modifica variabila originală. Acest mecanism funcţionează şi pentru structuri şi uniuni.

Excepţia apare la şiruri, deoarece la orice apel de funcţie cu argument şir, C-ul transmite adresa primului element din şir. Acest lucru explică redundanţa operatorului & în faţa numelui şirului. Transmiterea adresei şirului este eficientă la execuţie, dar, dacă funcţia modifică un element al şirului printr-un pointer, schimbarea afectează şirul original. În limbajul C nu există o cale directă de a crea o 'copie' a şirului pentru funcţia care îl utilizează. Există o metodă de a transmite întregul şir de elemente unei funcţii, metodă care permite modificarea elementelor şirului, fără ca aceasta să afecteze valorile originale ale elementelor.

Recomandăm această metodă la şiruri scurte, deoarece acest cert avantaj dispare la şirurile lungi datorită creşterii timpului de execuţie.

Această tehnică este utilizabilă şi la returnarea unui şir, la revenirea dintr-o funcţie.

4.5.1. Tehnica folosită pentru transferul şirurilor

Ca argument la funcţii se pot transmite şi şiruri întregi. Înainte de a transmite parametrul funcţiei, se construieşte un şir ca membru al unei structuri. Astfel, se poate transmite întregul şir funcţiei ca şi cum el ar fi o structură. Pentru exemplificare, fie programul ce manevrează un şir de *100* de întregi ca şi o structură.

```
struct buf{
        int s[ 100];
};

main()
{
        ...
        struct buf val;
        ...
        f( val);
        ...
}

f(struct buf s)
{
...
}
```


Se poate observa că această tehnică nu permite transmiterea şirurilor de dimensiuni diferite. Compilatorul trebuie să cunoască înainte de execuţia programului tipul şi dimensiunea şirului. Aceasta nu constituie un dezavantaj, întrucît oricum utilizăm această tehnică doar pentru şiruri de dimensiune mică.

Ilustrăm în continuare această tehnică prin două programe, descrise în 4.5.2 şi 4.5.3.

4.5.2. Şirul ca argument pentru funcţii

Presupunînd menţinerea unei liste cu numere de blocuri într-un şir, avem nevoie de o cale de a accesa datele din şir ca o listă de elemente sortate, fără însă a modifica lista iniţială. O soluţie este transmiterea întregului şir ca argument unei funcţii care va sorta elementele din copia şirului. La revenirea din funcţie, lista originală a numerelor de blocuri rămîne neschimbată.

Programul 4.8.

```
/* sort1.c */

#include <stdio.h>
#define MAX 9

struct buf {
        int val[ MAX];
};

void display( struct buf *ps, int nc)
{
        register int *p = ps->val;
        while ( p < ps->val + nc)
                printf("%4d", *p++);
        putchar('\n');
}

void sort( struct buf s, int nc)
{
        register int *p;
        int sortat = 0;
        while ( !sortat) {
                sortat = 1;
```

```
            for ( p=s.val+1; p < s.val+nc; p++)
                    if ( *(p-1) > *p) {
            /* interschimbă elemente alăturate */
                            *p ^ = *(p-1);
                            *(p-1) ^ = *p;
                            *p ^ = *(p-1);
                            sortat = 0;
                    }
        }
        display( &s, nc);
}

main()
{
        static struct buf bloc = { \
                34, 16, 12, 44, 17, 99, 11, 37, 19 };
        display( &bloc, MAX);
        sort( bloc, MAX);
        display( &bloc, MAX);
}
```

Execuția programului afișează:

```
34      16      12      44      17      99      11      37      19
11      12      16      17      19      34      37      44      99
34      16      12      44      17      99      11      37      19
```

Linia a doua conține șirul sortat, iar linia a treia demonstrează nemodificarea șirului original.

Am declarat o structură șablon ce conține un șir de 9 întregi. Încărcăm numerele de blocuri într-o structură de acest tip și afișăm cu ajutorul funcției *display()* șirul inițial. Urmează sortarea cu ajutorul funcției *sort()*, în care se afișează rezultatul sortării (linia a doua) și o nouă afișare după revenirea din funcția *sort()*.

Funcția *sort()* are ca prim argument o structura (cu șirul numerelor de blocuri), iar al doilea argument este numărul de blocuri ce se sortează. Acest lucru ne permite o sortare parțială a șirului (dacă dorim). Atragem încă o dată atenția asupra transmiterii unei structuri funcției *sort()*, și nu a adresei ei.

În ceea ce privește algoritmul de sortare, acesta este simplu; se parcurge șirul de numere cu ajutorul unui pointer *p* și se interschimbă perechi de elemente,

dacă primul element este mai mare decît al doilea. Tehnica de interschimbare a fost prezentată deja.

4.5.3. Funcții ce returnează șiruri

Nu există nici un motiv pentru a nu putea utiliza aceeași idee pentru returnarea unui șir la revenirea dintr-o funcție. Pentru exemplificare, modificăm în programul anterior funcția *sort()*, astfel încît să returneze șir. Codul devine:

Programul 4.9.

```
/* sort2.c */
#include <stdio.h>
#define MAX 9

struct buf {
        int val[ MAX];
};
void display( struct buf *ps, int nc)
{
        register int *p = ps->val;
        while ( p < ps->val + nc)
                printf("%4d", *p++);
        putchar('\n');
}
struct buf sort( struct buf s, int nc)
{
        register int *p;
        int sortat = 0;
        while ( !sortat) {
                sortat = 1;
                for ( p=s.val+1; p < s.val+nc; p++)
                        if ( *(p-1) > *p) {
                        /* interschimba elemente alaturate */
                                *p ^ = *(p-1);
                                *(p-1) ^ = *p;
                                *p ^ = *(p-1);
                                sortat = 0;
                        }
        }
```

```
        return s;
}
main()
{
        static struct buf bloc = { \
                        34, 16, 12, 44, 17, 99, 11, 37, 19 };
        struct buf sort();
        struct buf new;
        display( &bloc, MAX);
        new = sort( bloc, MAX);
        display( &new, MAX);
        display( &bloc, MAX);
}
```

Variabila *new*, care este o structură de tip *buf*, reține valoarea returnată de *sort()*, adică șirul sortat afișat de al doilea apel al funcției *display()*. Primul și ultimul apel al funcției *diplay()* afișează șirul original. Ieșirea generată la execuție rămîne aceeași.

Atragem încă o dată atenția asupra utilizării corecte a structurilor ca argumente pentru funcții. Dacă nu utilizăm pointeri, vom avea un timp adițional la transmiterea și la returnarea structurilor prin apelurile de funcții. Ca atare, transmitem adresa structurii funcției *display()*, deoarece ea, contrar funcției *sort()*, nu necesită modificarea datelor. La transmiterea/returnarea structurilor la/din funcții, compilatorul copiază datele originale.

4.6. Funcții cu număr variabil de argumente

Limbajul C, spre deosebire de multe limbaje, permite apelul unei funcții cu un număr variabil de argumente ce au tipuri diferite. Funcțiile *printf()* și *scanf()* sînt exemple concludente. Spre exemplificare, să considerăm un program ce citește datele din intrarea standard prin funcția *scanf()* și le afișează la ieșirea standard prin funcția *printf()*.

Programul 4.10

```
#include <stdio.h>
main()
{
        int num;
        float val;
```



```
        char nume[80];
        scanf("%d", &num);
        scanf("%f %s", &val, nume);
        printf("%f %s\n", val, nume);
        printf("%d %f %s\n", num, val, nume);
}
```

Apelul funcțiilor *scanf()* și *printf()* se face cu două argumente la primul apel și cu trei argumente la al doilea apel. Se observă că argumentele funcției *scanf()* sînt pointeri, pe cînd cei ai funcției *printf()* sînt de diverse tipuri.

Dacă dorim să scriem o funcție cu număr variabil de argumente, o soluție ar fi declararea acelei funcții cu un număr de argumente mai mare decît ar apărea la apelul ei și fixarea tipului argumentelor la *double* (limbajul C nu face conversie de tip implicită pentru argumentele sale, iar tipul *double* acoperă orice tip de variabilă utilizată).

Pentru exemplificare, considerăm un program ce afișează informații de depanare, în funcție de un anumit fanion:

Programul 4.11.

```
/* varg2.c - număr variabil de argumente */
#include <stdio.h>
int flags[] = { 0, 1, 1, 0};
main()
{
        int num;
        float val;
        char nume[ 25];
        printf("Introduceti un numar intreg: ");
        scanf(" %d", &num);
        printf("Introduceti un numar real si un sir de caractere: ");
        scanf(" %g  %s", &val, nume);
        debug( 1, "%d ", num);
        debug( 2, "%g %s\n", val, nume);
}
debug( int i, char *fmt, double arg1, double arg2, double arg3)
{
        if ( flags[ i])
                fprintf( stderr, fmt, arg1, arg2, arg3);
}
```

Şirul *flags* activează fanioanele de pe pozitiile 1 şi 2.

Afişarea informaţiei de depanare din interiorul funcţiei debug() se face doar dacă variabila *flags[i]* este poziţionată pe 1.

Acest program ilustrează posibilitatea apelului unei funcţii C cu un număr de argumente mai mic decît cel existent în declaraţia ei.

Program nu este portabil. El lucrează incorect dacă un compilator transmite argumentele pe stivă în altă ordine decît altul. Apare aşadar necesitatea unui program portabil.

4.6.1. Tehnica

Existenţa fişierului **stdarg.h** ne oferă interfaţa de care avem nevoie. Utilizînd declaraţiile din acest fişier, programul precedent devine:

```
/* varg3.c - număr variabil de argumente */

#include <stdio.h>
#include <stdarg.h>
int flags[] = { 0, 1, 1, 0};

main()
{
        int num;
        float val;
        char nume[ 25];
        printf("Introduceti un numar intreg: ");
        scanf(" %d", &num);
        printf("Introduceti un numar real si un sir de caractere: ");
        scanf(" %g  %s", &val, nume);
        debug( 1, "%d ", num);
        debug( 2, "%g %s\n", val, nume);
}
debug( int i, char *fmt, ...)
{
        va_list args;
        va_start( args, fmt);
        if ( flags[ i])
                vprintf( fmt, args);
        va_end( args);
}
```


Funcţia *debug()* are un număr variabil de argumente, motiv pentru care în definirea funcţiei există secvenţa "...".

Fişierul **stdarg.h** conţine declaraţia tipului *va_list* şi trei macrouri *va_start*, *va_arg* şi *va_end*. În documentaţia de Turbo C se găsesc declaraţiile:

```
#include <stdarg.h>

void va_start (va_list ap , lastfix);
type va_arg (va_list ap, type);
void va_end (va_list ap);
```

Unele funcţii, ca *vfprintf()* şi *vprintf()* lucrează cu număr variabil de parametri. Macrourile *va_...* reprezintă o modalitate de a accesa lista de argumente pas cu pas, fără a cunoaşte dinainte numărul de argumente din listă.

Pentru exemplul prezentat, declaraţia:

```
va_list args;
```

defineşte lista de argumente ce va fi utilizată de macrourile *va_arg* şi *va_end*.

Înainte de a descărca argumentele din stivă este necesară iniţializarea listei de argumente. Apelul:

```
va_start(args , fmt)
```

iniţializează lista de argumente *args* la primul argument fixat din lista variabilă, şi anume şirul *fmt* (*lastfix* din documentaţie). Apelul acesta este obligatoriu înainte de apelurile *va_arg()* sau *va_end()*.

Rutina *va_arg()* descarcă stiva de argumente. Primul argument al funcţiei este numele listei de argumente (*ap* din documentaţie), iar al doilea este tipul datei care se descarcă. Programul trebuie să conţină un nume de variabilă de acelaşi tip cu tipul datei ce se descarcă din stivă, pentru a putea memora valoarea returnată de *va_arg()*. Spre exemplu, instrucţiunile:

```
int inum;
char * fmt;
.
.
.
inum = va_arg(args, int);
fmt  = va_arg(args, char *);
```

memorează un întreg în variabila *inum*, declarată de tip *int*, și adresa unui șir în variabila *fmt*, declarată pointer la caracter. Analog se procedează și pentru structuri și uniuni. Dacă dorim să accesăm un tip *char*, e necesar să utilizăm tipul *int* în apelul rutinei *va_arg()*. Analog, pentru tipul *float* e necesar tipul *double* (se consideră tipul acoperitor). Următoarele instrucțiuni pun în evidență cele discutate:

```
char ch;
float fv;
.
.
.
ch = va_arg (args , int);
fv = va_arg (args , double);
```

Dacă nu se respectă aceste considerente, stiva se descarcă greșit și apar valori incorecte.

TC oferă posibilitatea utilizării unui set de funcții ce permit accesul la argumentele dintr-o listă, fără a specifica fiecare argument individual. Lista acestor funcții este :

```
/* afișare formatată a unei liste de argumente */
int vprintf (char *fmt,va_list args)

/* afișare formatată a unei liste de argumente într-un fișier */
int vfprintf (FILE *stream, char *fmt,va_list args)

/* afișare formatată a unei liste de argumente într-un șir */
int vsprintf (char *s, char *fmt, va_list args)
```

Aceste funcții sînt similare celor ce nu au litera '*v*' în față.

Funcția *debug()* din exemplul dat afișează informația de depanare prin funcția *vprintf()*, fără a mai utiliza rutina *va_arg()*.

Ultima rutină apelată este *va_end()*, care șterge lista de argumente.

4.6.2. Modul de lucru cu funcția va_arg()

Fișierul **stdarg.h** conține următoarele :


```
.
.
.
typedef void * va_list;
#define va_start( ap, parmN)(ap =...)
#define va_arg( ap,type)(*((type *)(ap))++)
#define va_end(ap)
#define va_ptr(...)
.
.
.
```

Prima linie definește tipul *va_list* ca pointer la *void*. Următoarele trei linii conțin macrodefiniții pentru funcțiile *va_start()*, *va_arg()* și *va_end()*. Bazîndu-ne pe cele discutate anterior, funcția *va_arg()* va accesa lista de argumente prin intermediul pointerului *ap*. Acesta este convertit la pointer la tipul *type* prin utilizarea castului:

```
((type *)(ap)).
```

Se aplică apoi operatorul de indirectare *, care va selecta primul argument din listă, iar apoi pointerul se incrementează. Astfel, la fiecare apel ulterior al funcției *va_arg()*, se va returna următorul argument din listă.

Funcțiile cu număr variabil de argumente trebuie să cunoască numărul de argumente ce se află pe stivă. Dacă nu se utilizează o funcție de bibliotecă asemenea funcției *vprintf()*, funcția trebuie să cunoască pe lîngă numărul de argumente și tipul fiecărui argument. Fără aceste informații funcția nu poate accesa individual argumentele.

Programul următor ilustrează un asemenea exemplu. Utilizînd un șir de formate (similar funcțiilor *scanf()* și *printf()*) funcția *myprintf()* cunoaște la apel numărul de argumente și tipul lor. Codul programului este:

Programul 4.11.

```
/* varg5.c - funcție cu număr variabil de argumente */
#include <stdio.h>
#include <stdarg.h>
main()
{
        static char buf[] = "demonstrație tc";
        int num = 44;
```

```
        char c = 'X';
        float f = 7.77;
        myprintf("%s", buf);
        myprintf("%s, %d", buf, num);
        myprintf("%s, %d, %c", buf, num, c);
        myprintf("%s, %d, %c, %f", buf, num, c, f);
        myprintf("%f %c, %d, %s", f, c, num, buf);
        myprintf("%x", num);
}

myprintf( char *fmt)
{
        va_list args;
        char *p;
        int inum;
        double fnum;
        char *ps;
        int ch;
        va_start( args, fmt);
        printf("\napel myprintf cu formatul %s\n", fmt);
        for( p = fmt; *p; p++) {
                if ( *p == '%')
                        switch ( p[1]) {
                                case 's':
                                        ps= va_arg( args, char *);
                                        printf(" şir: %s\n", ps);
                                        break;
                                case 'd':
                                        inum = va_arg( args, int);
                                        printf(" nr. întreg: %d\n", inum);
                                        break;
                                case 'c':
                                        ch = va_arg( args, int);
                                        printf(" caracter: %c\n", ch);
                                        break;
                                case 'f':
                                        fnum = va_arg( args, double);
                                        printf(" nr. real: %g\n", fnum);
                                        break;
                        default:
                                printf(" %c este un format ilegal\n", p[1]);
```



```
                                        break;
                                        }
        }
        va_end( args);
}
```

Se observă că şirul de format "*%x*" este nepermis în cazul acestei funcţii, deoarece el nu este tratat. Rezultatul afişat în urma execuţiei programului este:

apel myprintf cu formatul %s
şir: demonstraţie tc

apel myprintf cu formatul %s, %d
şir: demonstraţie tc
nr. întreg: 44

apel myprintf cu formatul %s, %d, %c
şir: demonstraţie tc
nr. întreg: 44
caracter: X

apel myprintf cu formatul %s, %d, %c, %f
şir: demonstraţie tc
nr. întreg: 44
caracter: X
nr. real: 7.77

apel myprintf cu formatul %f %c, %d, %s
nr. real: 7.77
caracter: X
nr. întreg: 44
şir: demonstraţie tc

apel myprintf cu formatul %x
x este un format ilegal

4.7. Probleme

1. Să se decidă care din expresiile de mai jos sînt portabile:

```
*p >= '0' && *p++ <= '9' || *p >= 'a' && *p++ <= 'f';
a[i]=i++ > 10 ? 10 : i;
f( ++*p, ++p);
g(( ++*p, ++p));
while ( ++argv, --argc)
```

2. Programul următor incrementează fiecare element al unui șir cu valoarea 1.

```
/* tabl.c - incrementez elementele unui tablou */
#include <stdio.h>

main()
{
        static int tbl[2][3] = {
                1, 2, 3,
                4, 5, 6
        };
        int *p;

        for ( p=(int *)tbl; p<(int *)tbl+6;)
                ++*p++;
        for ( p=(int *)tbl; p<(int *)tbl+6;)
                printf("%2d", *p++);
        putchar('\n');
}
```

Explicați ceea ce face instrucțiunea ++*p++. Este ea o l-valoare ?

3. Descifrați declarația:

```
char (*(*pf))()[10];
```

Sînt prea multe paranteze ? Declarația:

```
char * (*pf)()[10];
```

este compilată corect ?

4. Cum arată declarația unui șir *buf* de *20* de pointeri la funcții ce returnează *10* pointeri la *float*.

5. Utilizînd tehnica transferului rapid de șiruri să se scrie o versiune proprie a funcției de bibliotecă *strncpy()*. Pentru șiruri lungi care dintre ele este mai rapidă? De ce versiunea dumneavoastră nu e recomandată ?

6. Să se scrie o funcție cu număr variabil de argumente care aduna oricîte numere din linia de comandă.

7. Să se analizeze programul:

```
/* fșir.c - transfer rapid de șiruri */

#include <stdio.h>
#define MAX 80

main()
{
        struct chars {
                char buf[MAX];
        };
        char a[MAX];
        *( struct chars *)a = *( struct chars *)"corect";
        printf("%s\n", a);
}
```

Să se explice rezultatul execuției. Codul este portabil ?

8. Să se modifice programul **sort2.c** pentru a transmite funcției *sort()* un șir bidimensional. Pentru a ordona datele se folosește un pointer la întreg. Care este calea cea mai eficientă de-a transmite un șir multidimensional ca membru de structură ?

9. Să se scrie o versiune proprie a funcției *scanf()* pentru șiruri, numere întregi și flotante. Funcția va include verificări pe care funcția *scanf()* nu le face (caractere ilegale, pointeri *NULL*). De exemplu:

```
        int *pnum;
        myscanf("%d", pnum);
```

afișează un mesaj de eroare dacă *pnum* este *NULL*. Analog dacă *fval* este un caracter ilegal apelul:

```
        myscanf("%f", &fval);
```

afișează din nou un mesaj de eroare.

5. Tehnici de depanare

Efortul necesar pentru depanarea unui proiect soft este, de regulă, subestimat. Programatorii caută adeseori instrumente de depanare abia după terminarea proiectului, în loc să-şi structureze mai întîi, mental cel puţin, posibilităţiile de depanare cele mai potrivite.

Din păcate, posibilităţile vaste de depanare a proiectelor soft sînt minimalizate. Utilizarea unui instrument de depanare adecvat oferă trei avantaje. Primul ar consta în detecţia erorii înainte de apariţia ei. Al doilea oferă posibilitatea utilizării aceluiaşi instrument, atît în depanare, cît şi în întreţinerea şi menţinerea proiectului soft; iar ultimul garantează că orice alt instrument de depanare, utilizat pentru aceaşi aplicaţie, nu oferă mai multe informaţii utile de diagnosticare.

Vom numi aceasta *programare defensivă*. Chiar şi programele ce necesită viteză în execuţie pot îngloba cod de depanare. De regulă, cea mai bună abordare în depanare este combinarea unui instrument simplu de depanare cu depanatorul C favorit.

Prezentăm în cele ce urmează cîteva posibilităţi de depanare a programelor C. Sînt prezentate tehnici de depanare ce utilizează instrucţiuni preprocesor, sau funcţii de bibliotecă şi sau, opţiuni de compilare (din Turbo C 2.0).

5.1. Preprocesorul C

Instrucţiunile preprocesor permit instrucţiunilor de depanare să coexiste cu codul de bază (productiv). Includerea în codul programului a unor teste de interfaţă permite modulelor program autotestarea.

La compilarea unui program C, preprocesorul este primul care parcurge codul sursă al programului. Preprocesorul nu este parte integrantă a compilatorului C şi, ca atare, are sintaxă proprie. El interpretează doar liniile codului sursă care sînt precedate de simbolul #. Prezentăm în cele ce urmează doar instrucţiunile preprocesor relevante în depanare.

Instrucţiunea preprocesor:

```
#define NUME constantă
```

defineşte constanta simbolică *NUME* şi îi înlocuieşte textual fiecare apariţie (mai puţin cea inclusă între apostofuri) cu valoarea *constantă*. Orice instrucţiune C

poate folosi constanta simbolică *NUME* şi testa valoarea ei. Dacă valoarea *constantă* lipseşte, constanta se iniţializează cu valoarea 1. De exemplu:

```
#define MAX
```

defineşte constanta simbolică *MAX* ca avînd valoarea 1.

Compilarea unui program cu opţiunea *-D*, specificată în linia de comandă, permite definirea de simboluri preprocesor fără a fi necesară definirea lor în codul sursă. Mai utilizate sînt formatele de compilare:

```
c:\tc>tcc -DNUME arg
c:\tc>tcc -DNUME=constantă arg
```

Ambele formate definesc constanta simolică *NUME*, analog unei instrucţiuni *#define* în codul sursă.

Dacă se utilizează primul format, programul va utiliza în codul sursă instrucţiunea preprocesor *#ifdef*. Fie programul **pgm.c** ce conţine secvenţa de cod:

```
#ifdef DBUG
        printf("val= %d ptr=%x\n", val, p);
#endif
...
#ifdef DBUG
        printf( "apel de funcţie...");
#endif
```

Compilarea programului **pgm.c** cu opţiunea *-DDBUG* include în codul compilat instrucţiunile *printf()*. Fără această opţiune, ele nu ar fi compilate. Este bine să nu se utilizeze la afişare funcţia *printf()*, deoarece se combină ieşirea programului cu informaţiile de depanare. În locul ei se poate utiliza funcţia *fprintf()*.

Codul programului **pgm.c**, utilizînd funcţia *fprintf()*, devine:

```
#include <stdio.h>
...
#ifdef DBUG
        fprintf(stderr, "val= %d ptr=%x\n", val, p);
#endif
...
#ifdef DBUG
        fprintf(stderr, "apel de funcţie...");
#endif
```

În acest caz, afișarea informațiilor de depanare se va face în fișierul standard de erori (*stderr*). Comanda:

>pgm > fies

va redirecta rezultatul execuției programului în fișierul **fies**.

Formatul al doilea inițializează constanta *NUME* cu șirul *constantă*. Programul va utiliza în codul sursă instrucțiunea *#if* pentru testarea valorii constantei simbolice. Programul următor, **dumpbuf.c**, utilizează valoarea constantei simbolice *DBUG* pentru a afișa un șir în hexazecimal, octal sau zecimal.

Programul 5.1.

```
/* dumpbuf.c - afișează tamponul în zecimal, octal și hexa */

#include <stdio.h>

#define MAXBUF 35
#define DEC 0

main()
{
        unsigned int buf[MAXBUF];
        int i;
        for ( i=0; i<MAXBUF; i++)
                buf[i] = i + 1;
#if DBUG>0 && DBUG<3
        printa( buf, MAXBUF, DBUG);
#endif
        putchar('\n');
        printa( buf, MAXBUF, DEC);
}
printa( register unsigned int *pb, int nitem, int debug)
{
        register int i;
        static char *pfmt[3] = { "%4d%c", "%4x%c", "%4o%c"};
        for ( i=0; i<nitem; i++)
                printf( pfmt[ debug], *pb++, (i+1) % 10 ? ' ': '\n');
        putchar('\n');
}
```


Dacă valoarea constantei *DBUG* este mai mare ca zero și mai mică decît trei, se apelează funcția *printa()*. Funcției i se transmite valoarea acestei constante ca al treilea parametru. Ca atare, *printa()* utilizează direct *DBUG*, prin variabila locală *debug*, ceea ce-i permite funcției să existe într-un alt fișier, care nu se compilează cu opțiunea *-DDBUG*.

Funcția *printa()* parcurge tamponul *buf*, prin intermediul pointerului *pb*, afișînd cîte zece elemente pe fiecare linie. Variabila *debug* este utilizată pentru selectarea formatului de afișare (hexazecimal, zecimal sau octal).

Compilarea programului, din linia de comandă, sub forma:

```
>tcc -I\tc -L\tc -DDBUG=1 dumpbuf.c
```

afișează la execuție:

```
 1   2   3   4   5   6   7   8   9   a
 b   c   d   e   f  10  11  12  13  14
15  16  17  18  19  1a  1b  1c  1d  1e
1f  20  21  22  23

 1   2   3   4   5   6   7   8   9  10
11  12  13  14  15  16  17  18  19  20
21  22  23  24  25  26  27  28  29  39
31  32  33  34  35
```

Primele patru linii afișează conținutul tamponului în hexazecimal, iar ultimele patru în zecimal. Recompilarea și execuția programului cu valoarea lui *DBUG* egală cu doi afișează aceleași date, dar în octal.

```
>tcc -I\tc -L\tc -DDBUG=2 dumpbuf.c
>dumpbuf
```

```
 1   2   3   4   5   6   7  10  11  12
13  14  15  16  17  20  21  22  23  24
25  26  27  30  31  32  33  34  35  36
37  40  41  42  43

 1   2   3   4   5   6   7   8   9  10
11  12  13  14  15  16  17  18  19  20
21  22  23  24  25  26  27  28  29  39
31  32  33  34  35
```

Compilarea fără opțiunea de depanare afișează la execuție conținutul tamponului în zecimal:

```
>tcc -I\tc -L\tc dumpbuf.c
>dumpbuf
```

```
 1   2   3   4   5   6   7   8   9  10
11  12  13  14  15  16  17  18  19  20
21  22  23  24  25  26  27  28  29  39
31  32  33  34  35
```

Programul nu compilează condiționat funcția *printa()*. Acest lucru face posibilă utilizarea funcției și în module program care nu utilizează depanarea.
S-au folosit funcțiile *printf()* și *putchar()*, deoarece în toate cele trei situații se afișează conținutul tamponului în zecimal.

5.2. Teste de interfața înglobate

Opțiunea de compilare *-D* permite înglobarea așa-numitelor teste de interfață în codul sursă. Ele permit depanarea modulelor în momentul execuției și evitarea erorilor ce pot apărea ulterior. Includerea funcției principale *main()* într-o astfel de interfață ar putea fi:

Programul 5.2.

```
/* pgm.c - test de interfață */

#ifdef DRIVER
main( int argc, char *argv[])
{
        /* loc pentru cod de test*/
        func1();
        func2();
}
#endif

func1()
{
        /* loc pentru codul funcției */
}
```



```
func2()
{
        /* loc pentru codul funcției */
}
```

Compilarea și testarea programului **pgm.c** se realizează prin comanda:

```
>tcc -I\tc -L\tc -DDRIVER pgm.c
```

Furnizarea datelor de intrare pentru program se poate face dintr-un fișier sau de la tastatură. Dacă există certitudinea că funcțiile *func1()* și *func2()* funcționează corect, se recompilează programul cu opțiunea *-c*.

```
>tcc -I\tc -L\tc -c pgm.c
```

Rezultă fișierul obiect **pgm.obj**, care se poate lega (linkedita) împreună cu celelate module pentru a forma programul complet.

Un exemplu elocvent îl constituie modulul **hex.c**, care conține o funcție ce convertește un număr hexazecimal (dat sub formă de șir) într-un număr întreg. Pentru depanare s-a introdus un test de interfață.

Programul 5.3.

```
/* hex.c - conversie din hexa în întreg */

#include <stdio.h>
#include <ctype.h>

#ifdef DRIVER
main( int argc, char *argv[])
{
        unsigned hex();
        if ( argc == 1) {
                fprintf( stderr, "Utilizare: hex <număr>\n");
                exit(1);
        }
        printf("%u\n", hex( argv[1]));
}
#endif
```

```
ied hex( char *p)

        unsigned char digit;
        unsigned rez=0;
        for ( ; *p; p++) {
        if ( !isxdigit( *p)) {
                fprintf(stderr,"'%c' este un caracter hexa ilegal\n",*p);
                exit(1);
                }
                digit -= (( digit = tolower( *p)) <= '9') ? '0': 'a'-10;
                rez = ( rez << 4) + digit;
        }
        return rez;
}
```

Interfața permite introducerea de date test din linia de comandă. Dacă apelul funcției este incorect, se afișează un mesaj de informare care precizează formatul de utilizare. Funcția *printf()* afișează rezultatul conversiei șirului hexa citit ca argument din linia de comandă.

Funcția *hex()* conține instrucțiunile:

```
digit -= (( digit = tolower( *p)) <= '9') ? '0': 'a'-10;
rez = ( rez << 4) + digit;
```

ce utilizează o combinație de tehnici prezentate în capitolul 1.

Compilînd programul împreună cu interfața de depanare

```
>tcc -I\tc -L\tc -DDRIVER hex.c
```

obținem pentru execuții cu date diferite următoarele rezultate:

```
>hex
Utilizare: hex <număr>
>hex ff
255
>hex Ff
255
>hex abc
2748
>hex foc
'o' este un caracter hexa ilegal
```


Rezultatele obținute certifică corectitudinea programului, astfel încît programul se poate recompila cu opțiunea *-c*. În cazul unui rezultat posibil eronat se recompilează programul prin includerea interfației de depanare și se verifică execuția pe data care a generat rezultatul eronat.

5.3. Mesaje de informare în format ASCII și hexa

Un instrument simplu de depanare îl constituie un program care afișează conținutul unei zone de memorie, în hexazecimal și ASCII. Acest program va fi inclus în altele, cu scopul de-a pune în evidență cîteva tehnici de depanare.

Programul **adump.c** utilizează opțiunea *-D* de două ori la compilare. Prima testează programul **adump.c** cu un program de interfață, iar a doua generează o tabelă de caractere ASCII. Formatul liniei de comandă necesar la compilarea programului este:

```
>tcc -I\tc -L\tc -DDRIVER -DTEST adump
```

Programul necesită ca argument în linia de comandă un număr care reprezintă dimensiunea zonei de memorie afișate. În cazul în care el lipsește, se afișează mesajul de informare:

```
Utilizare: adump <dim>
```

Compilarea programului cu opțiunea *-D* și constanta *TEST* definită, generează la execuția cu argumentul *128* precizat în linia de comandă, următoarea tabelă de caractere ASCII.

```
>adump 128

000:    0001 0203 0405 0607 0809 0a0b 0c0d 0e0f  ................
016:    1011 1213 1415 1617 1819 1a1b 1c1d 1e1f  ................
032:    2021 2223 2425 2627 2829 2a2b 2c2d 2e2f   !"#$%&'()*+,-./
048:    3031 3233 3435 3637 3839 3a3b 3c3d 3e3f  0123456789:;<=>?
064:    4041 4243 4445 4647 4849 4a4b 4c4d 4e4f  @ABCDEFGHIJKLMNO
080:    5051 5253 5455 5657 5859 5a5b 5c5d 5e5f  PQRSTUVWXYZ[\]^_
096:    6061 6263 6465 6667 6869 6a6b 6c6d 6e6f  `abcdefghijklmno
112:    7071 7273 7475 7677 7879 7a7b 7c7d 7e7f  pqrstuvwxyz{|}~.
```

Datele afișate sînt împărțite pe trei coloane. În coloana din stînga este afișat, în zecimal, deplasamentul unei adrese de memorie. Coloana din mijloc conține

datele de la adresele respective în format hexazecimal, iar coloana a treia aceleași date, dar în cod ASCII. Programul **adump.c** afișează caracterul '.' pentru caracterele ASCII netipăribile.

Compilarea programului **adump.c** fără opțiunea *-DDTEST* produce o tabelă cu date arbitrare. Acest lucru confirmă abilitatea programului de-a afișa date arbitrare din memorie. Iată două exemple:

```
>tcc -DDRIVER adump
>adump 128

000:    245d 663e 5036 103a 6d43 6e3a 795f 1e65  $]f>P6.:mCn:y_.e
016:    147d 0e00 6e01 0867 7061 014f 297c 1e3b  .}..n..gpa.O)|.;
032:    7c3b 083b 0866 716c 5848 672d 4c07 581f  |;.;.fqlXHg-L.X.
048:    473d 6760 1a5b 6e07 3140 5363 7c15 1070 G=g'.[n.1@Sc|..p
064:    216c 7e1b 6019 6238 464f 3a40 157f 494f  !l~.'.b8FO:@..IO
080:    746e 605d 5315 6f3c 213d 4d53 325b 4419  tn']S.o<!=MS2[D
096:    6b2a 6154 3007 7a14 0f11 006a 2d01 066e  k*aT0.z....j-..n
112:    7146 1270 7066 247e 1911 0516 2207 522c  qF.ppf$~....".R,

>adump 77

000:    245d 663e 5036 103a 6d43 6e3a 795f 1e65  $]f>P6.:mCn:y_.e
016:    147d 0e00 6e01 0867 7061 014f 297c 1e3b  .}..n..gpa.O)|.;
032:    7c3b 083b 0866 716c 5848 672d 4c07 581f  |;.;.fqlXHg-L.X.
048:    473d 6760 1a5b 6e07 3140 5363 7c15 1070 G=g'.[n.1@Sc|..p
064:    216c 7e1b 6019 6238 464f 3a40 15         !l~.'.b8FO:@.
```

Prezentăm codul programului **adump.c** în două părți. Pentru început, programul de interfață.

Programul 5.4.

```
/* adump.c */

#include <stdio.h>
#define HEX 1
#define ASCII 2
#ifdef DRIVER
#ifndef TEST
#define TEST 0
#endif
```



```
main( int argc, char *argv[])
{
        char *buf;
        char *malloc();
        int buf_dim;

        if ( argc==1) {
                fprintf( stdout, "Utilizare: %s <dim>\n", argv[0]);
                exit(1);
        }

        buf_dim = atoi( argv[1]);
        buf = (char *) malloc( buf_dim);
        if ( buf == ( char *) NULL) {
                fprintf( stdout, "Nu există spațiu heap\n");
                exit(1);
        }

        umple( buf, buf_dim);
        adump( buf, buf_dim);
}

umple( char a[], int dim)
{
        register int i;
        srand( i);

        for ( i=0; i<dim; i++)
                a[i] = (TEST) ? i : rand() & 0x7f;
}
#endif
```

Programul de interfață constă din funcția principală *main()* și funcția *umple()*. Dacă constanta *TEST* nu este definită, programul de interfață o actualizează cu valoarea *0*. Se alocă apoi memorie heap pentru un tampon de dimensiune specificată (argumentul liniei de comandă) și se apelează funcția *umple()* pentru actualizarea tamponului cu date. Apelul funcției *adump()* afișează conținutul tamponului.

Funcția *umple()* generează numere funcție de valoarea constantei *TEST*. Dacă ea este zero, programul de interfață apelează funcția *rand()* pentru generarea unor numere aleatoare. În caz contrar, actualizarea tamponului se face

cu numere succesive. Se maschează valoarea returnată de funcția *rand()* cu *0x7f* pentru a memora în tampon fiecare număr ca un caracter ASCII.

A doua parte a programului conține funcțiile *adump()* și *putoct()*.

```
adump( char *p, int dim)
{
        register int i;
        register char *q = p;
        for ( i=0; i<dim; i += 16, q += 16) {
                fprintf( stdout, "\n%03d:\t", i);
                if ( i+16 <= dim)
                        putoct( q, 16, HEX);
                else
                        putoct( q, dim-i, HEX);
                if ( dim-i != 1)
                        fputc('\t', stdout);
                if ( i+16 <= dim)
                        putoct( q, 16, ASCII);
                else
                        putoct( q, dim-i, ASCII);
        }
        fputc('\n', stdout);
}

putoct( char *p, int n, int mod)
{
        register char *q = p;
        register int i;
        for ( i=0; i<n; i++, q++) {
                if ( mod == HEX) {
                        if ( i%2 == 0)
                                fputc(' ', stdout);
                        fprintf( stdout, "%02x", *q);
                        continue;
                }
                if ( mod == ASCII)
                        if ( *q < ' ' || *q > '~')
                                fputc('.', stdout);
                        else
                                fputc( *q, stdout);
        }
```



```
        if ( n != 16) {
                for ( i=0; i<16-n; i++) {
                        fputc(' ', stdout);
                        if ( mod == HEX) {
                                fputc(' ', stdout);
                                fputc(' ', stdout);
                        }
                }
        }
}
```

Ca și implementare, funcția *adump()* utilizează o buclă *for* pentru afișarea adreselor de memorie și apelează funcția *putoct()* pentru afișarea a cîte *16* octeți pe linie. Funcția *putoct()* afișează linia în format hexazecimal sau *ASCII* funcție de variabila *mod* (a cărei valoare poate fi *HEX* sau *ASCII*). După testare, programul se poate recompila fără programul de interfață. Programul compilat **adump.obj** se va utiliza în secțiunea următoare.

5.4. Fanioanele liniei de comandă

În cazul în care se dorește afișarea unor mesaje distincte, se impune frecvent utilizarea combinată a funcției *fprintf()* cu instrucțiuni de compilare condiționată. O variantă mai bună este apelul unei funcții de depanare care conține ca argument o variabilă fanion. Acest lucru permite afișări distincte funcție de valoarea fanionului. Spre exemplu, în loc de:

```
#include <stdio.h>
...
#ifdef DBUG
fprintf(stderr, "num = %u\n", num);
#endif
...
#ifdef DBUG
fprintf(stderr, "val[5] = %d\n", val[5]);
#endif
```

se poate utiliza:

```
#include <stdio.h>
#include "debug.h"
...
```

```
DEBUG(1,"num = %u\n", num);
...
DEBUG(2,"val[5] = %d\n", val[5]);
#endif
```

DEBUG fiind un apel macro definit în fișerul **debug.h**.
Fiecare apel macro *DEBUG* afișează valoarea unei variabile funcție de fanionul utilizat. Se pot utiliza atîtea fanioane cîte sînt definite în fișierul **debug.h**.

Presupunem programul **pgm.c** ce conține apelul macro *DEBUG*. Compilarea lui în forma:

```
>tcc -I\tc -L\tc -DDBUG pgm.c
```

activează depanarea. Pentru inițializarea unui fanion și afișarea informației de depanare corespunzătoare lui, se execută programul cu opțiunea *-D*. Spre exemplu comanda:

```
>pgm arg1 arg2 ... -D1
```

determină execuția programului **pgm.c** cu argumentele *arg1*, *arg2*, ... și inițializează fanionul de depanare 1. Ca atare, *DEBUG* afișează informația de depanare corespunzătoare fanionului 1. În această abordare fanionul apare la sfîrșitul liniei de comandă, după argumente. Există și alte metode pentru accesul la fanioane (a se vedea problema 3).

Programul necesită apelul unei funcții speciale *DEBUGFLAGS* pentru accesul la fanioanele din linia de comandă. Este necesar un șir pentru memorarea numerelor de fanion.

Să ilustrăm această tehnică pe problema calului de șah. Tabla pătratică, pe care se deplasează calul cu scopul de-a acoperi fiecare pătrat al ei, este de dimensiune variabilă. Dimensiunea laturii ei se transmite ca parametru din linia de comandă. Pentru o tablă de dimensiune dată, programul precizează dacă problema are sau nu soluție. Programul conține și cod de depanare. Funcție de fanionul precizat în linia de comandă, programul poate afișa soluția finală sau încercări de găsire a soluției. Pentru afișare programul utilizează funcția *adump()* definită în secțiunea anterioară. Formatul de afișare este același, prezentînd interes în acest caz doar coloana din mijloc. Ea afișează tabla, fiecare grup de patru cifre (doi octeți) reprezentînd în hexacimal numărul mutării prin care calul a ocupat pătratul. Un grup de patru cifre zero marchează un pătrat încă neocupat.

Execuția programului cu linia de comandă de forma:


```
>cal 25 -D1
```

afișează:

Problema are soluție
Problema calului pe tabla de latură: n=5

```
000:   0100 0600 0f00 0a00 1500 0000 0000 0000 ........
016:   0e00 0900 1400 0500 1000 0000 0000 0000 ........
032:   1300 0200 0700 1600 0b00 0000 0000 0000 ........
048:   0800 0d00 1800 1100 0400 0000 0000 0000 ........
064:   1900 1200 0300 0c00 1700 0000 0000 0000 ........
```

Se observă că fanionul 1 afișează prima soluție găsită.

Programul **cal.c** apelează macroul *DEBUGFLAGS* pentru citirea fanioanelor din linia de comandă și funcția *adump()* pentru afișarea tablei în hexazecimal. Codul programului este:

Program 5.5.

```
/* cal.c */

#include <stdio.h>
#include "debug.h"

int Debug[ NFLAGS];

int incerc(int,int,int);
void print(void);

# define LIBER 0
# define SUCCES 1
# define INSUCCES 0

static int a[8] = {2, 1, -1, -2, -2, -1, 1, 2};
static int b[8] = {1, 2, 2, 1, -1, -2, -2, -1};
int h[8][8];
int N, NP;
```

```
void main( int argc, char *argv[])
{
  int i,j;
  if ( argc < 2) {
        printf("Utilizare: cal <nr> [-D1][-D2]\n");
        exit(1);
  }

  N=atoi(argv[1]);
  NP=N*N;

  DEBUGFLAGS( &argc, argv); /* citesc fanioanele */
  DEBUG( 1, "Problema calului pe tabla de latură: n=%d\n", N);

  for (i=0; i<N; i++)
     for (j=0; j<N; j++);
        h[i][j] = LIBER;
  h[0][0] = 1;
  if (incerc(2,0,0) == SUCCES) {
     printf("Problema are soluție\n");
     if ( Debug[1])
        adump( &h[0][0], 2*N*8);
  } else
     printf("Problema nu are soluție\n");
}
int incerc(i,x,y)
{
  int k,xu,yu,q;
  for (k=0, q=INSUCCES; q==INSUCCES && k<8; k++) {
     xu=x+a[k];
     yu=y+b[k];
     if ( xu>=0 && xu<N && yu>=0 && yu<N)
        if ( h[xu][yu] == LIBER) {
           h[xu][yu]=i;
           if (i < NP) {
              if ((q=incerc(i+1,xu,yu)) == INSUCCES)
                 h[xu][yu] = LIBER;
           } else
              q = SUCCES;
        }
  }
```



```
  if (i==NP && Debug[2])
     adump( &h[0][0], 2*8*N);
  return(q);
}
```

În program se include fișierul **debug.h** și se declară șirul *Depan* pentru memorarea fanionelor. Numărul maxim de fanioane (*NFLAGS*) este definit în **debug.h**.

Comanda:

```
>tcc -I\tc -L\tc -DDBUG cal.c adump.obj
```

compilează și linkeditează, cu activarea depanării, programul **cal.c**.

Codul fișierului **debug.h** este:

```
/* debug.h */

#define NFLAGS 16

#ifdef DBUG
#define TRUE 1
extern int Debug[];
#define DEBUG( flag, fmt, arg) \
        if (Debug[flag]) \
                fprintf( stderr, fmt, arg)
#define DEBUGFLAGS( pac, av)\
        debugflags( pac, av)

static debugflags( int *pargc, char *argv[])
{
   int argc=*pargc;
   int dflag;
   while ( argc-- > 1) {
      if (strncmp( argv[argc], "-D", 2) ==0) {
         if (sscanf(&argv[argc][2],"%d", &dflag) != 1 ||
               dflag >= NFLAGS) {
            fprintf(stderr,"%s opțiune de depanare eronată\n",
               argv[argc]);
            exit(1);
         }
```

```
            Debug[dflag]=TRUE;
            argv[argc]=NULL;
            (*pargc)--;
        }
    }
}
#else
#define DEBUG(flag, fmt, arg)
#define DEBUGFLAGS(pac, av)
#endif
```

Se permit maxim 16 fanioane. Dacă programul **cal.c** este compilat fără opțiunea *-DDBUG* preprocesorul invalidează toate apelurile macro *DEBUG* și *DEBUGFLAGS*. În caz contrar, apelul macro *DEBUG* verifică fanionul corespunzător din șirul *Debug* și apelează funcția *fprintf()*, dacă fanionul a fost poziționat.

Funcția *debugflags()* citește fanionele din linia de comandă în ordine inversă. Cînd funcția întîlnește o opțiune *-D*, utilizează funcția *sscanf()* pentru a extrage numărul de fanion și îl inițializează cu *TRUE* în șirul *Debug*. Dacă numărul de fanion este un caracter ilegal, *sscanf()* returnează un mesaj de eroare și programul se termină.

Transmiterea adresei argumentului *argc* funcției *debugflags()* permite funcției să decrementeze contorul de argumente (*argc*) prin utilizarea expresiei compacte cu pointer:

```
(*pargc)--;
```

Prin această dispunere, programele ce apelează *DEBUGFLAGS* pot accesa argumentele liniei de comandă fără a cunoaște opțiunea *-D*. În linia de comandă, înaintea fanionelor, se acceptă oricîte argumente. Instrucțiunea:

```
argv[argc]==NULL;
```

introduce un pointer *NULL* la sfîrșitul listei de argumente.

Se poate include apelul macro *DEBUG* și în alte module. Pentru exemplificare să apelăm *DEBUG* din **adump.c**.

```
/* adump.c - afișare cod ASCII în hexa */
#include <stdio.h>
#include "debug.h"
...
```



```
adump( char *p, int dim)
{
        for ( i=0; i<dim; i += 16, q += 16) {
                DEBUG( 3, "\n%03d:\t", i);
                ...
}

putoct( char *p, int n, int mod)
{
....
        DEBUG( 4,"\n%03d:\t", n);
        if ( n != 16) {
                for ( i=0; i<16-n; i++) {
                ...
}
```

Programul **adump.c** include fișierul **debug.h**, dar nu trebuie să declare șirul *Debug*. Acesta este partajat de toate modulele ce utilizează macro apelul *DEBUG* și este declarat o singură dată. Informațiile de depanare pentru programul **cal.c** le putem afișa în cîteva moduri distincte. Spre exemplu comanda:

```
>cal 5 -D2
```

afișează toate încercările de acoperire a tablei, care eșuează la ultimul pătrat. Se observă pe cele cinci coloane, care formează tabla prezența unei singure grupări de zerouri.

```
000:    0100 0600 0b00 1600 1300 0000 0000 0000 ................
016:    0a00 1700 1400 0500 0c00 0000 0000 0000 ................
032:    0f00 0200 0700 1200 1500 0000 0000 0000 ................
048:    1800 0900 1000 0d00 0400 0000 0000 0000 ................
064:    0000 0e00 0300 0800 1100 0000 0000 0000 ................

000:    0100 0600 0b00 1600 1100 0000 0000 0000 ................
016:    0a00 1700 1000 0500 0c00 0000 0000 0000 ................
032:    0f00 0200 0700 1200 1500 0000 0000 0000 ................
048:    1800 0900 1400 0d00 0400 0000 0000 0000 ................
064:    0000 0e00 0300 0800 1300 0000 0000 0000 ................

...
```

```
000:    0100 0600 0f00 0a00 1700 0000 0000 0000 ..............
016:    0e00 0900 1800 0500 1000 0000 0000 0000 ..............
032:    1300 0200 0700 1600 0b00 0000 0000 0000 ..............
048:    0800 0d00 1400 1100 0400 0000 0000 0000 ..............
064:    0000 1200 0300 0c00 1500 0000 0000 0000 ..............

000:    0100 0600 0f00 0a00 1500 0000 0000 0000 ..............
016:    0e00 0900 1400 0500 1000 0000 0000 0000 ..............
032:    1300 0200 0700 1600 0b00 0000 0000 0000 ..............
048:    0800 0d00 1400 1100 0400 0000 0000 0000 ..............
064:    1900 1200 0300 0c00 1700 0000 0000 0000 ..............
```

Problema are soluție

Au fost afișate doar trei încercări și soluția finală.

Comanda:

```
>cal 5 -D1 -D2
```

afișează împreună informațiile de depanare corespunzătoare celor două fanioane. Execuția programului fără opțiuni afișează mesajul că problema are soluție.

5.5. Semnale și informații de depanare

Această secțiune prezintă modul în care execuția programelor poate fi intreruptă dacă se dorește afișarea unor mesaje de depanare. Metoda folosește semnale, care au fost discutate în capitolul 4.

Turbo C oferă în acest sens funcția *signal()* în vederea generării unui semnal. Semnalele pot proveni de la programul însuși sau de la alte programe. Orice program ce folosește semnale trebuie să includă fișierul **signal.h**. Formatul de utilizare este următorul:

```
#include <signal.h>

typedef void( *sigfun)( int subcod);
int signal( int sig, sigfun func);
```

Parametrul *sig* este un număr de semnal, de regulă o constantă simbolică a cărei valoare este precizată în fișierul **signal.h**, iar *func* este un pointer la o

funcție de tratare a semnalului de valoare *sig*. Funcția nu returnează nimic semnificativ, ca atare tipul ei este *void*.

Pentru depanare utilizăm un număr de semnal, pentru a cărui tratare trebuie scrisă o funcție. Pentru a ilustra acest lucru, presupunem un program C care așteaptă și captează un semnal definit anterior (o întrerupere). În momentul sosirii semnalului dorit este apelată funcția de tratare, care atribuie fanionului de control valoarea 1 și revine în programul principal.

În programul principal este testată valoarea fanionului, iar dacă aceasta este 1, se apelează o funcție ce afișează informațiile de depanare. Situația se repetă la fiecare apariție a întreruperii respective. Programul afișează informațiile de depanare în fișierul standard de erori, astfel încît rezultatul execuției programului se poate redirecta într-un fișier separat.

Fanionul de întrerupere *debug* este declarat static pentru a evita un eventual conflict de nume cu alte module program. Primul apel al funcției *signal()* din *main()* informează că semnalul de valoare *SIGINT* este întreruperea care se tratează. În cazul tastării combinației *Ctrl-C*, se apelează funcția *captez()*. Aceasta ignoră prin *SIG_IGN* alte semnale *SIGINT* și actualizează cu 1 fanionul de întrerupere.

Bucla infinită din *main()* testează fanionul de întrerupere. Dacă acesta are valoarea 1, programul afișează mesajul: "*semnal captat...*", urmat de afișarea conținutului unui tampon. În final se actualizează cu zero fanionul de întrerupere și întregul proces se repetă.

Demonstrativ, execuția programului **sig2.c** este ilustrată în cele ce urmează:

Programul 5.6.

```
/* sig2.c - semnale și informații de depanare */

#include <stdio.h>
#include <signal.h>
#define MAX 50
static int debug;

main()
{
  unsigned char buf[MAX];
  void captez();
  signal( SIGINT, captez);
  printf("\nUtilizare:Ctrl-C = se afișează conținutul tamponului\n");
  printf("========== O tasta = sfîrșit program\n");
```

```
    printf("executie...\n");
    while (1) {
      if ( kbhit()){
        raise(SIGTERM);
      }
      fillbuf( buf);
      if (debug) {
        fprintf( stderr,"semnal captat\n");
        display( buf);
        debug=0;
        signal( SIGINT, captez);
      }
    }
}

void captez()
{
  signal( SIGINT, SIG_IGN);
  debug=1;
}

void quit()
{
  exit(0);
}
fillbuf( unsigned char *p)
{
  register unsigned char *q=p;
  static unsigned char num;
  while ( q<p+MAX)
    *q++ = num++;
}
display( unsigned char *pb)
{
  register int i;
  for (i=0; i<MAX; i++)
    fprintf( stderr, "%4d%c", *pb++,(i+1) % 10 ? ' ':'\n');
  putc('\n', stderr);
  fflush( stderr);
}
```


Funcția *fillbuf()* inițializează tamponul *buf* cu valori întregi succesive, iar funcția *display()* afișează aceste valori, cîte zece pe fiecare linie. Inițializarea tamponului începe cu valoarea variabilei *num*, care fiind declarată *static* își păstrează valoarea de la un apel al altul al funcției *fillbuf()*. În consecință funcția inițializează tamponul cu date diferite la fiecare apel al ei din funcția *main()*. Rezultatul execuției programului se prezintă sub forma:

```
Utilizare: Ctrl-C = se afișează conținutul tamponului
========== O tastă = sfîrșit program
execuție...
^C
semnal captat
 160 161 162 163 164 165 166 167 168 169
 170 171 172 173 174 175 176 177 178 179
 180 181 182 183 184 185 186 187 188 189
 190 191 192 193 194 195 196 197 198 199
 200 201 202 203 204 205 206 207 208 209

^C
semnal captat
 10 11 12 13 14 15 16 17 18 19
 20 21 22 23 24 25 26 27 28 29
 30 31 32 33 34 35 36 37 38 39
 40 41 42 43 44 45 46 47 48 49

^C
semnal captat
 102 103 104 105 106 107 108 109 110 111
 112 113 114 115 116 117 118 119 120 121
 122 123 124 125 126 127 128 129 130 131
 132 133 134 135 136 137 138 139 140 141
 142 143 144 145 146 147 148 149 150 151
```

Conținutul tamponului este afișat la fiecare apăsare a combinației *Ctrl-C*. Programul se termină prin apăsarea unei taste.

5.6. Aserțiuni

Erorile de programare apar atît la compilare, cît și la execuție. Dacă erorile de compilare sînt ușor de depistat și, ca atare, de corectat, ele fiind

semnalate şi localizate de compilator, nu acelaşi lucru se poate spune despre erorile ivite la execuţie. Un program ce pare a lucra corect conform unei specificaţii de definiţie, poate genera rezultate neaşteptate la evenimente sau condiţii omise din specificaţie. Găsirea acestor erori necesită timp.

Secţiunea de faţă are menirea de a introduce tehnica aserţiunilor ca variantă de detectare a erorilor la execuţie. Aserţiunile sînt deosebit de utile în depanare, deoarece ele opresc, sau chiar termină, execuţia unui program înainte ca eroarea în execuţie să genereze un rezultat surpriză. Aserţiunea se poate aplica oricărui tip de dată, dar de regulă se foloseşte pentru şiruri şi pointeri. Prezentăm în cele ce urmează modul de lucru şi de utilizare a aserţiunilor.

Multe din erorile la execuţie au ca premiză rezultatul unei citiri sau scrieri de la sau la o locaţie greşită. De regulă, acest lucru se întîmplă în încercarea de referire a unui element care se află în afara şirului. Capitolul 3 a atras atenţia asupra nesemnalizării acestei erori la compilare.

Programul următor demonstrează acest lucru:

Program 5.7.

```
/* cîmp.c - referire ce depăşeşte marginea */

#include <stdio.h>

main()
{
        char buf[10];
        putchar( buf[-3] = 'x');
        putchar( buf[10] = 'y');
        putchar( buf[20] = 'z');
        putchar('\n');
}
```

Programul este compilat fără erori. Compilatorul converteşte referirea *buf[20]* la **(buf + 20)* şi programul referă o adresă de memorie care se află în afara zonei rezervate şirului *buf* (de zece octeţi). Acelaşi lucru se întîmplă şi pentru *buf[-3]*. Comportarea la execuţie a acestui program este funcţie de compilatorul utilizat (sub Turbo C versiunea 2.0 execuţia afişează şirul "*xyz*") .

Aserţiunea permite detecţia unei astfel de erori. Definiţia formală a unei aserţiuni este valoarea unei expresii boolene care în timpul execuţiei se aşteaptă să fie adevărată (*TRUE* = adevărat = nonzero). Dacă expresia este evaluată la fals (*FALSE* = fals = zero) programul se termină şi aserţiunea afişează un mesaj ce localizează eroarea în program.

TC implementează aserţiunile prin includerea fişierului **assert.h** în programul sursă.

```
#include <stdio.h>
#include <assert.h>
...
```

Fişierul **stdio.h** trebuie inclus, deoarece fişierul **assert.h** utilizează simboluri definite în el. Formatul unei aserţiuni este:

```
assert ( expresiebool)
assert ( expresiebool);
assert ( expresiebool) instrucţiune
```

Dacă *expresiebool* se evaluează la nonzero, aseţiunea este validă; în caz contrar, programul se termină cu un mesaj de eroare.
Cele trei formate de aserţiuni lucrează în acelaşi mod, astfel încît ele sînt echivalente. Caracterul ';' este opţional la terminarea unei aserţiuni. O aserţiune se poate plasa pe aceeaşi linie cu o instrucţiune program sau pe o linie separată.
Eliminarea aserţiunilor din programul sursă se face prin opţiunea de compilare *-D*.

Fie declaraţiile:

```
int num, index, buffer[MAX];
char c;
double *pheap;
```

Pentru ele sînt posibile aserţiunile:

```
...
assert ( c >= '0' && c <= '9')
num = c - '0';
...
assert ( index >= 0 && index < MAX);
buffer[index] = num;
...
assert( pheap != (double *) NULL) *pheap = 34.5;
```

Prima dintre ele verifică dacă variabila *c* reprezintă codul ASCII al unei cifre, înainte de conversia ei într-un număr întreg. Următoarea aserţiune iniţializează un element al şirului *buffer* cu valoarea lui *num*, nu înainte de-a verifică dacă indexul şirului, variabila *index*, depăşeşte marginea din stînga sau din dreapta a şirului. Ultima aserţiune verifică dacă pointerul *pheap* referă o locaţie

de memorie validă (diferită de *NULL*) pentru a memora date acolo. În cazul în care una dintre aceste aserțiuni eșuează, programul se termină.

Aserțiunile sînt utile, deoarece programele nu dețin întotdeauna controlul asupra valorilor pe care le poate lua o variabilă. Spre exemplu, *pheap* și *index* pot primi valori din linia de comandă, de la tastatură sau de la alte funcții. Fără aserțiuni, programele pot eșua sau pot avea execuții neașteptate.

Nu există reguli care să specifice cînd e indicat să se utilizeze aserțiuni. După cum am menționat, folosirea aserțiunii este indicată în cazurile în care nu se controlează domeniul de valori al unei variabile. Se recomandă utilizarea aserțiunilor în locuri în care este importantă o detectare precoce a unor erori ce ar cauza efecte dezastruoase. Pe de altă parte, nu trebuie utilizate aserțiuni pentru a înlocui verificările și mesajele de eroare. Aserțiunile sînt recomandate, mai degrabă, pentru situații ce nu ar trebui să apară niciodată. Aceasta cere experiență.

5.6.1. Aserțiuni pentru șiruri

Să considerăm un program demonstrativ ce folosește aserțiunea pentru a verifica argumentul liniei de comandă, care reprezintă un index într-un șir. De regulă, aserțiunile se utilizează pentru aplicații mult mai complexe.

Primul program, **șir1.c**, necesită două argumente în linia de comandă. Primul argument este un index într-un șir, iar al doilea este valoarea ce se va atribui elementului din șir care are acel index. Codul programului este:

Programul 5.8.

```
/* șir1.c - referire element din afara șirului */

#include <stdio.h>
#define MAX 10

main( int argc, char *argv[])
{
        int buf[MAX], index, val;

        if ( argc != 3) {
                fprintf(stderr,"Utilizare: %s index valoare\n", argv[0]);
                exit(1);
        }
        index = atoi( argv[1]);
```



```
        val = atoi( argv[2]);
        buf[index] = val;
        printf("buf[%d] = %d\n", index, buf[index]);
}
```

Acest program poate produce erori la execuție, avînd în vedere faptul că primul argument din linia de comandă (variabila *index*) poate fi orice număr. Execuția programului din linia de comandă:

```
c:\tc>șir1 12 99
```

afișează

buf[12] = 99

Aparent programul lucrează corect, eroarea la execuție nefiind semnalată. Programul următor este similar programului **șir1.c**, dar conține o aserțiune înglobată.

Programul 5.9.

```
/* șir2.c - referire element din afara șirului */
/*        varianta cu aserțiune */

#include <stdio.h>
#include <assert.h>

        ...
        val = atoi( argv[2]);

        assert( index >= 0 && index < MAX);

        buf[index] = val;
        ...}
```

Execuția acestui program, tot din linia de comandă, afișează:

Assertion failed: index >= 0 && index < MAX file șir2.c, line 20
Abnormal program termination

Mesajul de diagnosticare localizează aserțiunea eșuată în codul sursă al programului, prin afișarea numelui fișierului și a numărului liniei.

5.6.2. Aserțiuni pentru șiruri de caractere

Programul **string1.c** inițializează două șiruri de caractere și concatenează primul șir cu un șir primit ca argument din linia de comandă. Iată codul programului.

Programul 5.10.

```
/* string1.c */

#include <stdio.h>
#include <string.h>
#define MAX 15

main( int argc, char *argv[])
{
        static char s1[MAX] = "șir_test_";
        static char s2[MAX] = "alt_șir";

        if ( argc==1) {
                fprintf( stderr, "Utilizare: %s șir\n", argv[0]);
                exit(1);
        }
        printf(" s2 = %s\n", s2);
        strcat( s1, argv[1]);
        printf(" s1 = %s\n", s1);
        printf(" s2 = %s\n", s2);
}
```

Funcția *strcat()* plasează șirul primit din linia de comandă la coada șirului *s1*. În final, programul afișează șirurile *s1* și *s2*. Execuția programului pe o arhitectură cu 80286 va modifica șirul *s2*. Execuția programului se face din linia de comandă, iar rezultatul afișat este:

```
>string1 depășire
s2 = alt_șir
s1 = șir_test_depășire
s2 = re
```

Programul concatenează corect șirul *s1* cu șirul "*depășire*", dar scrie peste șirul *s2*, deoarece se depășește lungimea șirului *s1*. Iată așadar, un exemplu în care

o funcție de bibliotecă produce eroarea. Bineînțeles, funcția *strcat()* lucrează corect, deoarece ea lucrează cu pointer la caracter și nu cu șiruri de lungime dată.

O aserțiune bine plasată poate semnala din timp această eroare. Programul devine:

Programul 5.11.

```
/* string2.c - cu aserțiune */

#include <stdio.h>
#include <string.h>
#include <assert.h>

...
        printf(" s2 = %s\n", s2);

        assert(strlen(argv[1])<MAX-strlen(s1));

        strcat( s1, argv[1]);
...}
```

Execuția acestui program din linia de comandă afișează:

```
s2 = alt_șir
Assertion failed: strlen(argv[1])<MAX-strlen(s1) file string2.c; line 19
Abnormal program termination
```

5.6.3. Aserțiuni pentru pointeri

Aserțiunile pot testa și valida adrese de pointeri. Multe din erorile la execuție sînt cauzate de adrese de pointeri incorecte sau de pointeri neinițializați.

Ilustrăm cîteva exemple de aserțiuni ce testează valoarea pointerilor.

O aserțiune poate verifica valoarea returnată de o funcție. Presupunem, de exemplu, alocarea dinamică a unei zone de memorie pentru un bloc de întregi. Următorul segment de cod rezervă memorie heap pentru două șiruri de întregi.

```
int *pheap;
...
pheap = (int *) malloc( 100 * sizeof(int));
assert( pheap != (int *) NULL);
```

```
...
pheap = (int *) malloc( 50 * sizeof(int));
assert( pheap != (int *) NULL);
....
```

Pentru a fi siguri că alocarea a reuşit, s-a utilizat o aserţiune ce testează valoarea pointerului *pheap*. Dacă nu ar exista spaţiu suficient în memoria heap, *pheap* ar avea valoarea *NULL* şi programul s-ar termina. Acest lucru este testabil la fiecare încercare de alocare, pentru a putea identifica alocarea care dă greş.

Se pot utiliza aserţiuni şi în funcţii pentru testarea pointerilor la şiruri. Spre exemplu, o aserţiune verifică dacă valoarea unui pointer este o adresă cuprinsă în interiorul zonei pe care acest pointer o parcurge. Fie un şir *fval* de tip *double* şi un pointer *pd* care parcurge acest şir. Aserţiunea folosită în acest caz se va include în corpul funcţiei care lucrează cu pointerul.

```
#define MAX 100
double fval[ MAX];
...

func( double *pd)
{
        assert( pd >= fval && pd < fval + MAX);
        ...
}
```

În interiorul funcţiei *func()*, pointerul *pd* referă un element al şirului *fval*. Aserţiunea verifică dacă valoarea lui *pd* este o adresă validă, ce referă un element al şirului. Ea trebuie să fie cuprinsă în intervalul închis dat de adresa primului element din şir (numele şirului) şi adresa ultimului element din şir. Locul plasării aserţiunii este înaintea utilizării pointerului în funcţia *func()*. Şirul *fval* şi constanta simbolică *MAX* pot fi utilizate în aserţiune, deoarece ele sînt definite în afara funcţiei.

O aserţiune este utilă şi la validarea pointerilor dintr-o listă înlănţuită. Presupunem un program ce conţine următoarele declaraţii în fişierul **defs.h**.

```
struct nod {
        int data;
        struct nod *stg, *drp;
};
struct nod *pst, *psf;
```


Aceste declaraţii definesc o listă dublu înlănţuită. Fiecare nod conţine un cîmp de informaţie, materializat prin întregul *data* şi doi pointeri, unul la elementul precedent *pstg*, iar altul la elementul anterior *pdrp*. Această dispunere permite accesul rapid la nodurile listei, întrucît parcurgerea listei, începînd cu orice nod, este posibilă în ambele direcţii. Lista are definiţi (după modul în care s-a construit) doi pointeri. Primul, *pst*, marchează începutul listei, iar *psf* marchează sfîrşitul listei.

Pentru a pune în evidenţă aserţiunea, construim o funcţie ce necesită parcurgerea unei liste circulare. Fie aceasta funcţia *display_list()*, ce afişează cîmpul de informaţie (*data*) al unei liste de noduri. Lista de noduri este delimitată cu doi pointeri, unul ce marchează începutul, iar altul ce marchează sfîrşitul ei. Ca atare, apelul:

```
        display_list( pst, psf);
```

va afişa cîmpul de informaţie pentru toate nodurile listei.

Funcţia traversează o listă de noduri ce începe cu nodul referit de primul argument, pînă la întîlnirea nodului referit de al doilea argument din apel. Funcţia verifică dacă lista este corect construită (consistentă). Altfel spus, se verifică dacă pointerul spre dreapta al unui nod referă un nod al cărui pointer spre stînga este chiar nodul respectiv. Analog şi spre stînga. `Testul de consistenóÑ`, în acest caz, este realizat prin intermediul unei aserţiuni.

Codul funcţiei *display_list()* este:

```
#include <stdio.h>
#include <assert.h>
#include "defs.h"

display_list( struct nod *start, struct nod *end)
{
        struct nod *purm;

        for ( purm = start; ; purm = purm->drp) {
                assert( purm == purm->stg->drp);
                assert( purm == purm->drp->stg);
                printf("data = %d\n", purm->data);
                if ( purm == end)
                        return;
        }
}
```

Am utilizat pointerul *purm* pentru parcurgerea listei dublu înlănțuite. Cele două aserțiuni realizează testul de consistență înainte de utilizarea pointerilor. Expresia:

```
purm->drp->stg
```

se evaluează de la stînga la dreapta, nefiind necesare paranteze.

5.6.4. Eliminarea aserțiunilor din programul sursă

Deși aserțiunile *assert()* par a fi funcții, ele sînt implementate ca macrouri. Aceasta implică creșterea dimensiunii codului programului. Odată ce codul sursă a fost testat, el devine stabil, iar aserțiunile trebuiesc eliminate.
Opțiunea de compilare *-D* permite ștergerea aserțiunilor. Formatul de utilizare este:

```
c:\tc>tcc -DNDEBUG ...
```

unde *NDEBUG* este un simbol special definit în **assert.h**. Ca atare, o aserțiune este un segment de cod compilat condiționat, cînd simbolul preprocesor *NDEBUG* nu este definit. Din acest motiv, compilarea fără opțiunea *-D* implică compilarea liniilor ce conțin aserțiuni.

5.6.5. Modul de lucru al unei aserțiuni

Dacă aserțiunea este implementată ca macrou, ea este vizibilă în fișierul **assert.h**.

```
1       /*      assert.h

...
5       #include <stdio.h> /* fprintf() prototype
6               & 'stderr' definition */
7       #include <stdlib.h> /* abort() prototype */
8
9       #if !defined(NDEBUG)
10      #define assert(p) \
11              if(!(p)){ \
12      fprintf(stderr,"Assertion failed: %s, file\
13      %s, line %d\n",#p, __FILE__,__LINE__);\
```



```
14      abort();\
15      }\
16      #else
17      #define assert(p)
18      #endif
```

Numerele de linie nu apar în fișerul de mai sus, ele servind doar pentru referire. Pentru a vedea cum se șterg aserțiunile, să analizăm fișierul.

Presupunem compilarea unui program cu opțiunea *-DNDEBUG*, pentru eliminarea aserțiunilor. Linia *9* testează prin instrucțiunea preprocesor *#if* dacă simbolul preprocesor *NDEBUG* este definit. Dacă el este definit, preprocesorul trece la compilarea liniei *16* (ramura *else*). Instrucțiunea preprocesor din linia *17* determină înlocuirea în codul sursă a tuturor apelurilor macro *assert()* cu codul vid (nimic). Altfel spus, se șterg aceste linii din program. După analiza preprocesorului, compilatorul nu mai "vede" apelurile macro *assert()*.

La compilarea programului cu includerea aserțiunilor, constanta simbolică *NDEBUG* nu este definită, motiv pentru care aserțiunea se definește conform liniei *10*. Preprocesorul substituie fiecare apel macro *assert()* cu instrucțiunea *if* prezentă în liniile *11-16*. Esențială este substituția expresiei *p* cu expresia corespunzătoare din aserțiunea programului. După recompilare, compilatorul va compila instrucțiunea *if* ca și cum ea ar fi parte integrantă a codului sursă.

La execuția programului, instrucțiunea *if* evaluează expresia booleană substituită. Dacă rezultatul evaluării este zero (*FALSE*), se afișează în fișierul standard de erori un mesaj de diagnosticare și programul se termină prin apelul fucției *abort()*. În caz contrar, nu se face nimic. Ca exemplu pentru cele discutate anterior, aserțiunea:

```
assert( i >= 0 && i < MAX) data[i] = 10;
```

este substituită de preprocesor cu:

```
if ( !(i >= 0 && i < 10)) {
        fprintf(stderr,"Assertion failed: %s, file
        %s, line %d\n",#p, __FILE__,__LINE__);
        abort();
} else data[i] = 10;
```

în cazul compilării cu includerea aserțiunilor.

Constantele predefinite *__FILE__* și *__LINE__* sînt definite de preprocesor, ele putînd fi utilizate de orice program utilizator. Constanta

__FILE__ este un şir de caractere ce identifică numele fişierului, iar constanta *__LINE__* este un întreg ce identifică numărul liniei curente din fişier. Preprocesorul atribuie acestor constante valori înainte de compilare.

Pentru a ne convinge de aceste lucruri, fie programul de test **flinie.c**.

Programul 5.12.

```
/* flinie.c - */

main()
{
        printf("Acesta este fişierul %s\n", __FILE__);
        printf("linia = %d\n", __LINE__);
        printf("linia = %d\n", __LINE__);
        printf("linia = %d\n", __LINE__);
        printf("linia = %d\n", __LINE__);
}
```

Rezultatul execuţiei:

```
Acesta este fişierul FLINIE.C
linia = 7
linia = 8
linia = 9
linia = 10
```

Programul afişează numele fişierului şi cîteva numere de linii. Se observă cum preprocesorul substituie în fiecare linie valorea lui *__LINE__*.

Vom utiliza aceste constante predefinite într-un macrou propriu ce implementează o aserţiune.

5.6.6. Implementarea aserţiunilor proprii

S-ar putea ca mesajele de diagnosticare afişate de aserţiuni să nu fie satisfacătoare. În acest caz, înlocuim macroul ce implementează aserţiunea cu unul propriu. Fişierul care conţine definirea macro (macroexpandarea) îl numim tot **assert.h**. În acest caz, însă, includerea acestui fişier în programul principal este de maniera:

```
#include "assert.h"
```


Acest lucru realizează includerea fişerului propriu şi nu a celui standard.

Codul pentru acest fişier ar putea fi:

```
1       /* assert.h - propriu */
2
3       #ifndef NDEBUG
4       #define assert(expr) \
5               if (!(expr)) { \
6               fprintf(stderr,"Aserţiunea: %s",#expr);\
7               fprintf(stderr,"fişier %s, linia %d\n",
                        __FILE__,__LINE__);\
8               exit(1);\
9               }
10      #else
11      #define assert(expr)
12      #endif
```

Expresia *#expr*, utilizată şi în macroul standard, determină substituţia expresiei *expr* cu şirul de caractere ce materializează această expresie în momentul respectiv. Spre exemplu:

```
assert( ptr != NULL);
```

se expandează la:

```
fprintf(stderr,"Aserţiunea: %s","ptr != NULL");\
```

după substituţia preprocesor.

Utilizînd macroul propriu, programul **şir2.c** se modifică după cum urmează.

Programul 5.13.

```
/* şir3.c - şir2.c cu aserţiune proprie */

#include <stdio.h>
#include <stdlib.h>
#include "assert.h"
#define MAX 10
```

```
main( int argc, char *argv[])
{
        int buf[ MAX], index, val;
        if ( argc != 3) {
                fprintf( stderr, "Utilizare: %s index val\n", argv[0]);
                exit(1);
        }
        index = atoi( argv[1]);
        val = atoi( argv[2]);
        assert( index >= 0 && index < MAX);
        buf[ index] = val;
        printf("buf[%d] = %d\n", index, buf[index]);
}
```

Rezultatul execuției programului **şir3.c** este:

Aserțiune: index >= 0 && index < 10, fișier SIR3.C, linia 20

În locul funcției care determină terminarea programului, se poate apela o funcție utilizator. Această abordare oferă programatorului trei alternative: continuarea execuției, dacă gravitatea erorii nu impune terminarea, apelul automat al unui depanator sau afișarea unei ferestre cu mesaje de eroare.

Noua versiune pentru **assert.h** este:

```
/* assert1.h - fişier utilizator */

void (*assertf)(char *, char *, int);
#ifndef NDEBUG
#define assert( expr) if(!(expr)){\
        (*assertf)( #expr, __FILE__, __LINE__);\
}
#else
#define assert( expr)
#endif
```

Structura fișierului este similară cazului anterior. Modificarea apare în macro definiția lui *assert()*. Se definește pointerul la funcție, *assertf*, care se inițializează cu o funcție utilizator.

Funcția necesită trei argumente:
- un șir de caractere ce reprezintă aserțiunea,
- un pointer de tip char la numele fișierului,
- un întreg ce reprezintă numărul liniei.

Programul următor testează corectitudinea fișierului **assert1.h**. El citește caractere de la intrarea standard și le memorează într-un tampon. Dacă se citește un caracter invalid (o cifră), aserțiunea apelează funcția *adump()*, definită în secțiunile anterioare. Codul programului este:

Programul 5.14.

```
/*  assertf.c - aserțiune cu funcție utilizator */

#include <stdio.h>
#include "assert1.h"
#define MAX 100

char buf[ MAX];
void dump();

void (* assertf)();
main()
{
        int c, index=0;
        assertf = dump;
        while ( (c=getchar()) != '\n') {
                assert( c >= 'a' && c <= 'z');
                buf[index++] = c;
        }
}

void dump( char *passert, char *f, int nrlin)
{
        fprintf( stderr, "Eşec aserțiune: %s, ", passert);
        fprintf( stderr, "fişier %s, linie %d\n", f, nrlin);
        adump( buf, MAX);
        exit(1);
}
```

Funcția *dump()* este funcția utilizator de depanare. Programul atribuie pointerului la funcție (*assertf*) adresa funcției *dump()*. Dacă aserțiunea eșuează, apelul macro *assert()* apelează funcția *dump()*. Fiecărui caracter citit i se aplică aserțiunea pentru a vedea dacă acesta este un caracter literă mică. Dacă un caracter citit nu este literă mică, se afișează un mesaj de eroare, se apelează funcția *adump()* și programul se termină.

Pentru compilarea acestui program din mediul TC, avînd în vedere că funcţia *adump()* există deja în fişierul **adump.c**, editǎm un fişier proiect ce conţine liniile:

```
/* assertf.prj */
adump.obj
assertf.c
```

Execuţia lui **assertf.prj** cu argumentul "*exemplificare*" în linia de comandă afişează:

```
Eşec aserţiune: c >= 'a' && c <= 'z', fişier ASSERTF.C, linie 19

000:    6578 656d 706c 6966 6963 6172 6500 0000 exemplificare...
016:    0000 0000 0000 0000 0000 0000 0000 0000 ................
032:    0000 0000 0000 0000 0000 0000 0000 0000 ................
048:    0000 0000 0000 0000 0000 0000 0000 0000 ................
064:    0000 0000 0000 0000 0000 0000 0000 0000 ................
080:    0000 0000 0000 0000 0000 0000 0000 0000 ................
096:    0000 0000                               ....
```

Funcţia *dump()* afişează aserţiunea eşuată, numele fişierului, numărul liniei şi conţinutul tamponului.

5.7. Afişarea selectivă a informaţiilor de depanare

Toate tehnicile prezentate pînă în acest moment necesită recompilare la afişarea informaţiilor de depanare. Din acest motiv, în proiectele mari se consumă mult timp. Tehnica următoare permite afişarea selectivă a informaţiilor de depanare, fără a mai fi necesară recompilarea.

Această abordare este similară cu cea a argumentelor din linia de comandă. Apelurile funcţiei de depanare conţin fanioane ca argumente şi nu pot afişa tipuri de date compuse (şiruri, structuri sau uniuni). Un singur apel poate afişa oricîte variabile. Acest lucru nu este posibil prin macrouri, ci doar prin funcţii, deoarece numărul de argumente este variabil.

Vom crea un fişier de fanione în locul actualizării fanionelor din linia de comandă.

Există cîteva detalii de implementare. Primul apel al funcţiei de depanare caută în directorul curent fişierul de fanioane. Dacă el nu există, programul nu mai execută celelalte apeluri ale funcţiei de depanare. Dacă fişerul conţine

numere de fanioane, ele sînt memorate într-un şir de fanioane. Apeluri succesive ale funcţiei de depanare afişează informaţii de depanare corespunzătoare fanioanelor.

Programele ce necesită viteză la execuţie nu tolerează însă lucrul cu fişiere de fanioane.

Pentru a ilustra tehnica, se consideră programul **ddbug.c**.

Programul 5.15.

```
/* ddbug.c

Testare: tcc -I\tc -L\tc -DDRIVER ddbug.c
Utilizare:  tcc -c ddbug.c

*/

#include <stdio.h>
#include <assert.h>
#include <stdarg.h>
#include "dbug.h"

enum BOOL { FALSE, TRUE };
#define MAXF 50
int NrFlag;
#ifdef DRIVER
main()
{
        struct Persoana {
                char nume[15];
                char prenume[15];
                unsigned int cod;
                int vîrsta;
                double greut;
                double înalt;
                char ten[10];
        } X = { "Ionescu", "Ion", 101, 34, 72.45, 1.80, "brunet"};

        DBUG(1, "nume = %s\n", X.nume);
        DBUG(2, "prenume = %s\n", X.prenume);
        DBUG(3, "cod = %d\n", X.cod);
```

```
        DBUG(4, "vîrsta = %d\n", X.vîrsta);
        DBUG(5, "greutatea = %.7g\n", X.greut);
        DBUG(6, "înălțimea = %.3g\n", X.înalt);
        DBUG(7, "tenul = %s\n", X.ten);
        DBUG(8, "nume: %s prenume: %s\n",X.nume, X.prenume);
        DBUG(9, "nume: %s, %d ani, %.7g kg, %.3g m, ten %s \n",
                X.nume, X.vîrsta, X.greut, X.înalt, X.ten);
        if ( DBUG( 10, "Start...\n"))
                printf("...Sfîrșit\n");
        if ( DBUG(11, ""))
                printf("... sfîrșitul depanării\n");
}
#endif
```

Comentariul de la începutul programului prezintă formatul pentru compilarea din linia de comandă.
Programul include fișierul **assert.h**, care conține macroul pentru aserțiune și încă două fișiere pe care le vom prezenta pe parcurs.

Programul de interfață inițializează cîmpurile unei variabile *X*, de tip structură, cu niște valori arbitrare. Apelurile *DBUG* conțin ca argumente un număr de fanion, un format de afișare și variabila care se afișează. Dacă numărul de fanion există în fișierul de fanioane, mesajul de depanare este afișat în fișierul standard de erori. În caz contrar, nu se întîmplă nimic. Atîta timp cît *DBUG* este o funcție, valoarea returnată de ea se poate utiliza ca expresie booleană în orice instrucțiune C. *DBUG* returnează numărul de fanion, dacă el există, și zero, în caz contar.

Compilarea în formatul descris, în lipsa fișierului de fanioane (**flags.c**) nu afișează nimic. Dacă fișierul **flags.c** există și are conținutul:

1 2 3 4 5

execuția programului afișează:

nume: Ionescu
prenume: Ion
cod: 101
vîrsta: 34
greutatea: 72.45

Atragem din nou atenția că nu am recompilat programul. Dacă modificăm fișierul de fanioane, ieșirea se modifică funcție de numerele de fanione existente în fișier. Pentru:

1 2 3 4 5
8 9 10 11

obținem la execuție:

nume = Ionescu
prenume = Ion
cod = 101
vîrsta = 34
greutatea = 72.45
nume: Ionescu prenume: Ion
nume: Ionescu, 34 ani, 72.45 kg, 1.8 m, ten brunet
Start...
...Sfîrșit
.. sfîrșitul depanării

Se observă că fanionele 8 și 9 afișează mai mult de o variabilă.

Fanioanele 10 și 11 utilizează *DBUG* ca expresie booleană. Ambele apeluri se găsesc în testul unei instrucțiuni *if* și se evaluează la *TRUE* dacă fișierul de fanioane conține numerele 10 și 11. Apelul cu fanionul 10 afișează *Start...* urmat pe linia următoare de *...Sfîrșit*. Acest format este util la afișarea antetului de depanare la o afișare multiplă de mesaje de depanare. Spre exemplu, dacă fanionul 14 există, instrucțiunile:

```
if ( DBUG( 14,"tabela_de_simboluri\n"))
        dump_simb();
```

afișează șirul *tabela_de_simboluri* înainte de-a afișa simbolurile prin funcția *dump_simb()*.

Apelul *DBUG* cu 11 este util în execuția unui bloc de instrucțiuni. De exemplu, codul:

```
if ( DBUG( 15, "") {
        int c;
        adump( buf, MAX);
        printf("exit? ");
        c = tolower( getchar());
```

```
        if ( c == 'd')
                exit(1);
}
```

apelează funcția *adump()* pentru afișarea conținutului șirului *buf*, după care programul se poate termina dacă se tastează caracterul *'d'*. Instrucțiunile din bloc se execută doar dacă există fanionul 15.

Pentru eliminarea mesajelor de depanare, se șterge fișierul de fanioane.

Fișierul **dbug.h** are următorul conținut:

```
/* dbug.h */

extern NrFlag;
#define DBUG !NrFlag && dbug
```

Cele două instrucțiuni realizează nerepetarea apelului *DBUG* dacă nu există fișier de fanioane **flags.c**. În lipsa fișierului se inițializează variabila *NrFlag* cu valoarea *TRUE*. Preprocesorul substituie apoi apelul *DBUG* cu o instrucțiune ce testează *NrFlag*, înainte de apelul funcției *dbug()*. Spre exemplificare:

```
DBUG(5, "greutatea = %.7g\n", X.greut);
```

se substituie cu:

```
!NrFlag && dbug(5, "greutatea = %.7g\n",X.greut);
```

în care apelul *dbug()* nu se face dacă *NrFlag* este *TRUE*.

Următoarele linii din fișierul **ddbug.c** implementează funcția *dbug()*.

```
int dbug( int flag, ...)
{
        va_list args;
        char *fmt;
        FILE *fp;
        int f;
        static char readf = TRUE;
        static char flags[ MAXF];
        if ( readf) {
                if (( fp = fopen("flags.c", "r")) == NULL) {
                        NrFlag = TRUE;
                        return 0;
                }
```



```
                while ( fscanf( fp, "%d", &f) != EOF) {
                        assert( f>0 && f < MAXF);
                        flags[f] = TRUE;
                }
                fclose( fp);
                readf = FALSE;
        }
        va_start( args, int);
        fmt = va_arg( args, char *);
        if ( flags[ flag])
                vfprintf( stderr, fmt, args);
        va_end( args);
        return flags[ flag] ? flag : 0;
}
```

Funcția citește fișierul de fanioane și afișează mesajele de depanare. Din motive de portabilitate, funcția *dbug()* a fost implementată ca având un număr variabil de argumente. Tehnica aceasta a fost prezentată în capitolul 4.

Șirul *flags[]* memorează fanioanele active, iar variabila statică *readf* permite citirea fișierului de fanioane o singură dată.

Funcția de bibliotecă *fscanf()* citește fișierul de fanioane într-o buclă și verifică numărul de fanion prin intermediul unei aserțiuni. Fanionul *0* nu este disponibil, deoarece această valoare este returnată de funcție în cazul în care un fanion nu există.

5.8. Observații asupra funcțiilor de gestionare a memoriei

Un program care utilizează funcțiile de alocare fără verificări suplimentare este greu de depanat. Mesajele de eroare afișate de funcțiile *malloc()*, *calloc()* și *realloc()* sînt, de regulă, insuficiente pentru găsirea erorii. Distrugerea integrității memoriei heap, printr-un apel *free()*, cu un argument ce nu este o adresă rezultată dintr-un apel anterior al unei funcții de alocare, este un alt exemplu de eroare greu detectabilă.

5.8.1. Tehnica

Se completează funcțiile de alocare și eliberare cu verificări de eroare. Programele ce utilizează aceste funcții vor include fișierul **xalloc.h**. Funcția *xmalloc()* este varianta completată a funcției *malloc()*. Se presupune segmentul de cod:

```
#include "xalloc.h"
...
char *pheap;
...
pheap = (char *) xmalloc( 100);
...
```

Funcția *xmalloc()* alocă dinamic *100* de octeți. Dacă alocarea nu reușește din lipsă de spațiu, programul se termină și se afișează un mesaj de eroare care localizează în fișier apelul de alocare nereușit.

Apelurile macro *xrealloc()* și *xfree()* vor fi tratate în aceași manieră.

Conținutul fișierului **xalloc.h** este:

```
/* xalloc.h */

#define xmalloc(N)          ymalloc(__FILE__,__LINE__,N);
#define xcalloc(N,S)        ycalloc(__FILE__,__LINE__,N,S);
#define xrealloc(P,N)   yrealloc( __FILE__,__LINE__,P,N);
#define xfree(P)        yfree(__FILE__,__LINE__,P);

void *ymalloc(), *ycalloc(), *realloc();
void yfree();
```

Se observă implementarea apelurilor prin macrouri, ce apelează la rîndul lor funcții. Ca atare, un apel *xmalloc()* (macrou) apelează *ymalloc()* (funcție). Codul funcției *ymalloc()* este:

```
/* ymalloc1.c - rutina pentru malloc() */
#include <stdio.h>
void *
ymalloc(char *file, int lin, unsigned int noct)
{
        char *pheap; void *malloc();
        if ((pheap = (char *) malloc( noct)) == ( char *) NULL) {
                fprintf(stderr,"fișier %s-linia %d: eroare malloc
                        pentru %u octeți\n",file,lin,noct);
                exit(1);
        }
        return pheap;
}
```


Se apelează funcția *malloc()* și se verifică valoarea pointerului returnat de ea. Dacă valoarea pointerului este *NULL* (alocarea a eșuat), se afișează un mesaj de eroare și programul se termină.

Să considerăm un exemplu ce utilizează apelul *xmalloc()*. Fie un program ce gestionează o tabelă de simboluri. Pentru simplitate, limităm tipul datelor la tipul șir de caractere și la tipul *double*. O cale de a gestiona date de tipuri diverse este prin intermediul unei uniuni. Fișierul **defs.h** definește tipul *Symbol*. Tabela este implementată ca listă simplu înlănțuită de elemente de tip *Symbol*.

```
/* defs.h - definirea tabelei de simboluri */

#define SIR             1
#define DOUBLE          2

typedef struct Simbol {
        int dtip;
        union {
                char *pșir;
                double *pdouble;
        } val;
        struct Simbol *pnext;
} Symbol;
```

Programul utilizează apelul *xmalloc()* pentru a rezerva spațiu heap pentru două elemente de tip *Symbol* și pentru datele corespunzătoare lor. Ultimul apel *xmalloc()* prezintă o posibilă eroare. Codul programului și rezultatul execuției sînt ilustrate mai jos.

Programul 5.16.

```
/* simbol1.c - tabelă de simboluri */

/* Se va compila și executa fișierul proiect simbol1.prj, care conține liniile:
        ymalloc1.c
        simbol1.c
*/
#include <stdio.h>
#include "xalloc.h"
#include "defs.h"
```

```
main()
{
        Symbol *p1, *p2;
        char *ps = "testare";
        int *p5;
        p1=(Symbol *) xmalloc( sizeof( struct Symbol));
        p1->dtip = SIR;
        p1->val.psir = xmalloc( strlen(ps) + 1);
        strcpy( p1->val.psir, ps);
        p2=(Symbol*) xmalloc( sizeof( struct Symbol));
        p2->dtip = DOUBLE;
        p2->val.pdouble = ( double *) xmalloc( sizeof(double));
        *p2->val.pdouble = 6.7e-3;
        printf("%s\n", p1->val.psir);
        printf("%g\n", *p2->val.pdouble);
        p5 = (int *) xmalloc(0 * sizeof(int));
}
```

>simboll

testare

0.0067

fişier simboll.c - linia 26: eroare malloc pentru 0 octeţi

Primul apel *xmalloc()* alocă spaţiu heap pentru un obiect de tip *Symbol*, adică atîţia octeţi cît necesită cîmpurile structurii. Adresa acestui obiect este atribuită pointerului *p1* (pentru a putea referi apoi zona respectivă). Instrucţiunea ce urmează actualizează cîmpul *dtip* cu constanta simbolică *STRING*. Prin testarea acestui cîmp se stabileşte tipul simbolului. Al doilea apel *xmalloc()* alocă spaţiul necesar pentru şir, adică atîţia octeţi cîte caractere conţine şirul la care se adună octetul de sfîrşit de şir. Adresa returnată se atribuie cîmpului *psir*. Urmează apelul funcţiei *strcpy()*, care copiază şirul referit de *ps* în zona heap referită de *psir*.

Al treilea apel *xmalloc()* este identic cu primul, iar al patrulea alocă cei opt octeţi necesari pentru memorarea unui *double*. Se iniţializează cîmpul *dtip* cu constanta simbolică *DOUBLE*, iar în locaţia referită de cîmpul *pdouble* se memorează un număr real în dublă precizie (*6.7e-13*).

Ultimul apel *xmalloc()* eşuează.

O metodă eficientă de eliminare a erorilor la alocare-dealocare o constituie salvarea pointerilor alocaţi prin apelurile *xmalloc()* şi *xrealloc()*, astfel ca ei să poată fi testaţi de *xfree()* înaintea dealocării cu *free()*. Această măsură de precauţie evită dealocarea unui pointer heap invalid.

Pentru salvarea pointerilor folosim un şir *dbuf* de maximum 256 de pointeri.

```
#define MAXBUF 256
static char *dbuf[ MAXBUF];
```

Noua versiune pentru funcţia ymalloc() este:

```
/* ymalloc2.c - versiunea 2 */
#include <stdio.h>
void *
ymalloc( char *file, int nrlin, unsigned int noct)
{
        char *pheap; void *malloc();
        void install();
        if ( (pheap= (char *)malloc( noct)) == ( char *)NULL) {
                fprintf( stderr, "fişier %s - linie %d: eroare malloc
                        la alocarea %u octeţi\n",file,nrlin,noct);
                exit(1);
        }
        install(pheap);
        return pheap;
}
```

Singura diferenţă faţă de prima versiune este apelul funcţiei install(), care realizează salvarea pointerilor heap. Codul funcţiei *install()*:

```
/* memorează pointerii heap în şirul buf */

void install( char *pheap)
{
        register char **pbuf;

        for ( pbuf=dbuf; pbuf<dbuf+MAXBUF; pbuf++)
                if ( *pbuf == (char *) NULL) {
                        *pbuf = pheap;
                        return;
                }
        fprintf(stderr, "Spaţiu insuficient în şirul de alocare\n");
        exit(1);
}
```

Funcția *install()* memorează fiecare pointer heap rezultat după o operație de alocare, într-un element liber (*NULL*) al șirului. Dacă nu se găsește un asemenea element, se afișează un mesaj de eroare ce precizează că șirul este plin și programul se termină.

Căutarea unui element liber este liniară, dar sînt posibile și alte variante (Problemele 5 și 6).

Funcția *yrealloc()* verifică dacă pointerul primit ca argument, a fost anterior alocat. Dacă șirul de pointeri conține pointerul, se apelează funcția *realloc()* pentru realocarea unui nou pointer. Codul funcției *yrealloc()* este:

```
void
yrealloc( char *file, int nrlin, char *oldp, unsigned int noct)
{
        char *newp; void *realloc();
        register char **pbuf;
        int f=0;

        if ( oldp != ( char *) NULL)
                for ( pbuf=dbuf; pbuf<dbuf+MAXBUF; pbuf++)
                        if ( *pbuf == oldp) {
                                f=1;
                                break;
                        }
        if ( !f) {
                fprintf(stderr,"fișier %s - linie %d: eroare realloc
                        la adresa %x\n",file,nrlin,oldp);
                exit(1);
        }
        newp = (char *)realloc( oldp, noct);
        if ( newp == (char *) NULL) {
                fprintf(stderr,"fișier %s - linie %d: eroare realloc
                        pentru %u octeți %x\n",file,nrlin,noct);
                exit(1);
        }
        *pbuf = newp;
        return newp;
}
```

Inițial se testează dacă pointerul *oldp* este diferit de *NULL* și dacă există în șir. Dacă nu sînt respectate aceste condiții, se afișează un mesaj de eroare și programul se termină. Altfel, se apelează funcția *realloc()* ce testează noul

pointer *newp*. Înainte de-a returna pointerul, acesta se memorează în șirul de pointeri alocați.

Ultimul apel *xfree()* apelează funcția *yfree()*, ce verifică dacă un pointer corespunde unei alocări. Acest lucru este omis de funcția de bibliotecă *free()*. Codul funcției *yfree()* este:

```
void yfree( char *file, char *pheap, int nrlin)
{
        void free();
        register char **pbuf;

        if ( pheap != (char *) NULL)
                for ( pbuf=dbuf; pbuf<dbuf+MAXBUF; pbuf++)
                        if ( *pbuf == pheap) {
                                *pbuf = NULL;
                                free( pheap);
                                return;
                        }
        fprintf(stderr,"fișier %s - linie %d: eroare free
                la adresa %x\n",file, rlin,pheap);
        exit(1);
}
```

Funcția *yfree()* afișează un mesaj de eroare, care precizează adresa greșită și termină programul dacă pointerul heap este *NULL* sau dacă nu este memorat în șirul *dbuf*.

Completăm programul ce gestiona tabela de simboluri cu un apel *xrealloc()*, necesar pentru alocarea unui șir de caractere mai lung decît cel inițial, și cu un apel *xfree()*. Codul și rezultatul execuției programului **simbol2.c** sînt ilustrate mai jos.

Programul 5.17.

```
/* simbol2.c */
/* Se va compila și executa fișierul proiect simbol2.prj care conține liniile:
        ymalloc2.c
        simbol2.c
*/
#include <stdio.h>
#include "xalloc.h"
#include "defs.h"
```

```
main()
{
        Symbol *p1, *p2;
        char *ps = "şir de test";
        char *ps2 =  "mai lung şir de test";
        p1 = (Symbol *) xmalloc( sizeof( struct Symbol));
        p1->dtip = SIR;
        p1->val.pşir = xmalloc( strlen(ps) + 1);
        strcpy( p1->val.pşir, ps);
        p2 = (Symbol *) xmalloc( sizeof( struct Symbol));
        p2->dtip = DOUBLE;
        p2->val.pdouble = ( double *) xmalloc( sizeof(double));
        *p2->val.pdouble = 6.7e-3;
        printf("%s\n", p1->val.pşir);
        printf("%g\n", *p2->val.pdouble);
        p1->val.pşir = xrealloc( p1->val.pşir, strlen(ps2)+1);
        strcpy(p1->val.pşir, ps2);
        printf("%s\n", p1->val.pşir);
        xfree((char *) p2->val.pdouble);
        xfree(ps2);
}
```

```
>simbol2
```

şir de test
0.0067
fişier SIMBOL2.C - linie 2622: eroare free la adresa 20

Prima parte a programului este identică cu versiunea anterioară. Apelul *xrealloc()* cu argument vechiul pointer (*p1->val.pşir*), măreşte spaţiul alocat anterior, pentru a-l potrivi cu lungimea noului şir (păstrat de *ps2*). Programul copiază şirul în simbolul referit de *p1* şi afişează noul şir.

Programul conţine două apeluri *xfree()*, primul corect iar al doilea incorect. Apelul secund afişează un mesaj de eroare, deoarece *ps2* nu este un pointer heap. .

Performanţa programelor scade prin verificările de eroare introduse. Ca atare, se face un compromis între viteza de execuţie şi necesitatea verificărilor de eroare.

Avantajul rutinelor implementate este considerabil în activitatea de depanare. După ce codul este depanat, se pot şterge rutinele implementate, revenindu-se la funcţiile alocatorul standard. O cale simplă care realizează acest lucru este prin modificarea fişierului **xalloc.h**.


```
/* xalloc.h - fişier pentru alocatorul standard */

#define xmalloc(N)      malloc(N)
#define xcalloc(N, S)   calloc( N, S)
#define xrealloc(P, N)  realloc(P, N)
#define xfree(P)        free(P)

void *malloc(), *calloc(), *realloc();
void free();
```

Prin includerea acestui fişier în programul dorit, se vor apela funcţiile de bibliotecă C în locul rutinelor implementate.

5.9. Probleme

1. Să se modifice programul **adump.c**, astfel încît afişarea conţinutului unei zone de memorie să se facă în octal sau hexazecimal.
2. Să se modifice macroul *DEBUG* de la fanionele liniei de comandă pentru ca el să admită un număr variabil de argumente.
3. Să se modifice *DEBUGFLAGS*, astfel încît opţiunile liniei de comandă să poată fi tratate în orice ordine.
4. Să se scrie funcţia *xcalloc()* utilizînd tehnica prezentată la funcţiile *xmalloc()* şi *xrealloc()*.
5. La funcţiile de gestionare a memoriei heap s-a folosit şirul *dbuf* pentru a memora pointerii alocaţi. Să se modifice acest program, astfel ca dimensiunea şirului să nu fie fixă. Se va utiliza *malloc()* pentru a crea şirul şi *realloc()* pentru a mări dimensiunea lui, dacă este necesar.
6. În locul şirului *dbuf*, să se implementeze o listă înlănţuită care să gestioneze pointerii alocaţi. Să se compare, ca timp de execuţie, această soluţie cu cea de la problema 5.
7. Să se scrie un macrou care să permită schimbarea funcţiilor standard de gestionare a memoriei cu unele proprii, şi invers.

6. Alocator de memorie orientat pe obiect

6.1. Considerații teoretice

Funcțiile de biblioteca *malloc()*, *calloc()*, *realloc()* și *free()* oferă programului C controlul asupra alocatorului standard de memorie. Aceste funcții furnizează soluții portabile programelor ce gestionează memoria în timpul execuției lor. Ideea ce stă la baza alocatorului standard de memorie este de a aloca si elibera blocuri din memoria heap. Aceste blocuri de memorie pot avea dimensiunile cuprinse între un octet și limita superioara impusă de partea hard sau de sistemul de operare. Unitatea de alocare a memoriei heap este octetul. Programele pot lucra cu obiecte de dimensiuni mai mari prin conversia unui pointer de la tipul *char* (1 octet), la tipul de dată al obiectului utilizat. Acest aranjament permite programelor impunerea unei structuri superioare celei ce operează la nivel de șir de octeți.

Alocatorul standard de memorie are menirea de a ascunde programelor detaliile de implementare. În consecința, programatorii nu se preocupă de modul în care alocatorul gestionează memoria heap. Din această cauză, programatorii s-ar putea să utilizeze incorect funcțiile de alocare a memoriei. Indiciile unor astfel de probleme sînt datele eronate și epuizarea neașteptată a spațiului din memoria heap. Cel mai frecvent, aceste incorectitudini sau funcții eronate apar datorită programelor care alocă și eliberează repetat blocuri din memoria heap. Problemele apărute în timpul lucrului cu memoria heap sînt depistate mai greu, deoarece indiciile lor nu sînt mereu aceleași și adesea depind de factori nesemnalați.

Analizînd alocatorul standard, se poate formula următoarea constatare:

Adresa returnată în urma apelului funcțiilor *malloc()*, *calloc()* sau *realloc()* nu poate fi modificată de către alocatorul standard.

Acest fapt pare lipsit de importanță cît timp nu se sesizează că el împiedică efectiv alocatorul standard să rearanjeze memoria heap. În realitate, funcția *realloc()* apelează funcția *malloc()* și copiază datele, dacă este necesar, într-o nouă zonă a memoriei heap.

Fie un program utilizator care alocă si eliberează frecvent blocuri de memorie de dimensiuni diferite din memoria heap. După un timp memoria devine fragmentată. În consecință, apar zone libere (goale) între blocurile de date alocate. Din acest motiv, se poate întîmpla ca alocatorul standard să nu poată aloca un bloc din memoria heap, cu toate că aceasta poate conține suficienți octeți liberi (dar nu într-o zonă compactă).

Dacă alocatorul poate compacta spațiile prin rearanjarea datelor, tehnică denumită colectarea reziduurilor, el va putea satisface cererile ulterioare de alocare a memoriei heap și va evita astfel eșuarea programelor utilizator.
Evident, alocatorul nu poate rearanja datele dacă pointerii la date sînt modificați.

În capitolul de față este prezentat alocatorul standard într-o aplicație în care utilizarea sa pare corespunzătoare. Aplicația care o sugerăm este impementarea unui alocator de memorie orientat pe obiecte care să asigure și colectarea reziduurilor. Pentru rezolvarea acestei probleme furnizăm structura și codul sursă al unui alocator de memorie orientat pe obiecte. Acesta permite programelor utilizator să aloce și să elibereze obiecte de orice tip și dimensiune din memoria heap. Interfața sa este similară cu cea a alocatorului standard, cu deosebirea că funcțiile primului returnează un pointer la obiect în loc de pointer la void (a se vedea documentația Tc). Acest aranjament permite alocatorului de obiecte să rearanjeze memoria heap cînd aceasta este fragmentată. Colectarea reziduurilor devine automată și transparentă programelor utilizator. Cea mai mare parte a acestui capitol prezintă implementarea detaliată a alocatorului pe obiecte și analizează avantajele și dezavantajele sale.

6.1.1. Colectarea reziduurilor

Șirurile dinamice sînt acele șiruri care se modifică în timpul execuției programului. Nu ne interesează în mod special în ce fel se modifică conținutul șirurilor, ci doar atunci cînd modificarea dimensiunilor lor afectează șirurile adiacente. De exemplu, atunci cînd un șir devine mai lung, datele trebuie redistribuite pentru a se obține spațiul necesar. Pentru aceasta, se compactează datele existente, astfel încît spațiul liber să formeze o zonă contiguă ce poate fi reutilizată. Reamintim că alocatorul standard nu poate face acest lucru.

Această carență a alocatorului standard în colectarea reziduurilor nu este un neajuns de concepție. Alocatorul standard este conceput spre a fi un ansamblu portabil de funcții cu scopuri bine determinate. Dacă un programator trebuie să aloce memorie în mod dinamic, să redimensioneze sau să elibereze mici blocuri de memorie în mod frecvent, el va gestiona memoria heap cu un alocator distinct. Noul alocator păstrează evidența zonelor de memorie alocate, găsește zonele nealocate și compactează datele heap, dacă este necesar.

6.2. Alocatorul de obiecte

Alocatorul de obiecte gestionează obiecte din memoria heap. Obiectul este o structură cu cinci cîmpuri, care au semnificațiile:

- pointer la data asociată obiectului,
- dimensiunea (în octeți) a zonei alocate datei,
- dimensiunea (în octeți) ocupată efectiv de dată,
- pointer la obiectul următor,
- pointer la obiectul anterior.

Atît obiectul cît și data asociată se găsesc în memoria heap. Programele utilizator referă datele prin intermediul unui pointer la obiect. Acest aranjament permite alocatorului de obiecte să rearanjeze în orice moment memoria heap, fără a deranja execuția programului.

Există situații în care este necesară, din anumite motive, modificarea pointerului la data unui obiect. Dacă un program utilizator modifică dimensiunea datei unui obiect, alocatorul de obiecte poate modifica adresa heap. Colectarea reziduurilor poate modifica, de asemenea, adresele obiectelor.

Prin eliberarea unui obiect se eliberează structura asociată obiectului, ea devenind disponibilă pentru reutilizare.

Se prezintă în cele ce urmează, într-o manieră descendentă, codul sursă al alocatorului de obiecte. Se începe prin descrierea structurilor ce controlează alocatorul urmînd ca apoi să se descrie funcțiile de interfață ale alocatorului.

6.2.1. Obiecte și variabile de control

Alocatorul de obiecte controlează două zone separate din memoria heap: zona descriptorilor obiect și zona datelor obiect. Prima zonă conține informații care localizează și controlează datele asociate obiectelor, iar a doua conține efectiv datele.

Zona descriptorilor obiect este organizată sub forma unui șir de elemente de tipul *object*. Definirea tipului se află în fișierul **object.h**.

```
typedef struct Object {
        unsigned char *pobj;
        int lung;
        int util;
        struct Object *prec;
        struct Object *next;
} Object;
```

Un descriptor de obiect este un element de tipul *object*. Pointerul *pobj* accesează adresa de început (sau de bază) a datelor asociate descriptorului obiect. Cîmpul *lung* contorizează numărul de octeți alocați pentru date în zona datelor obiect, iar cîmpul *util* păstrează evidența numărului de octeți din zona de date pe

care îi utilizează efectiv obiectul. Motivul păstrării separate a variabilelor *lung* și *util* se va prezenta mai tîrziu, la fel și justificarea faptului că valoarea *util* variază între *0* și valoarea cîmpului *lung*. Alocatorul de obiecte nu incrementează variabila *lung* fără a cerceta unde este delimitat următorul obiect în memorie. De aceea, datele obiectului nu se suprapun niciodată.

Cu toate că descriptorii obiect sînt memorați într-un șir, descriptorii alocați formează o listă ordonată, dublu înlănțuită. Ordinea este dată de valoarea lui *pobj*, care păstrează adresa datei obiectului. Figura 6-1 ilustrează aceasta.

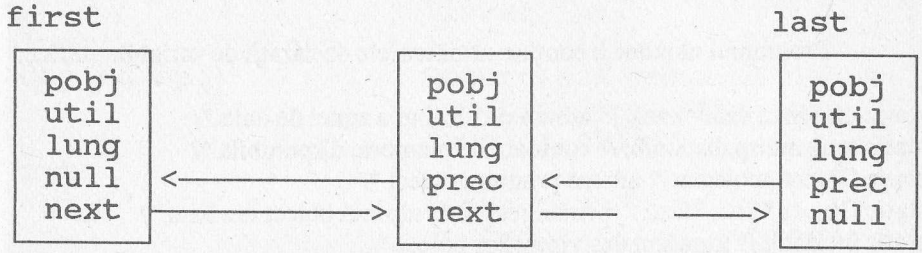


Figura 6.1. Lista descriptorilor obiect

Cîmpul *prec* și cîmpul *next* referă descriptorul obiect din stînga respectiv din dreapta descriptorului obiect curent. Alocatorul de obiecte parcurge lista descriptorilor obiect prin intermediul pointerilor *prec* (parcurge de la adrese superioare la adrese inferioare) sau *next* (parcurge de la adrese inferioare la adrese superioare.

Programul **objalloc.c** trebuie să conțină următoare directive:

```
#include <stdio.h>
#include "object.h"
#include "dbug.h"

#define OBJDIM 0X7000
#define OBJSPAȚIU 512
```

Fișierul **object.h** conține definiția tipului *object*, iar fișierul **dbug.h** conține liniile de cod:

```
extern Noflags;
#define DBUG !Noflags && dbug
```

S-a introdus apelul *DBUG* pentru ca execuţia programului să afişeze adresele pointerilor şi valorile anumitor variabile. La sfîrşitul capitolului se vor prezenta informaţiile de depanare afişate de un program utilizator.

Constanta simbolică *OBJSPAŢIU* reprezintă dimensiunea memoriei heap alocate pentru zona datelor obiect, iar constanta simbolică *OBJDIM* reprezintă numărul de descriptori obiect. Acesta este numărul maxim de obiecte cu care poate lucra alocatorul. În această implementare prezentă se folosesc 7000 hexa, adică 28672 octeţi pentru zona datelor obiect, şi 512 descriptori de obiecte. Aplicaţiile şi memoria disponibilă vor afecta aceste numere.

Programul **objalloc.h** conţine următoarele declaraţii de variabile statice:

```
static unsigned char *baza; /* adresa de început a zonei de date */
static long mdisp,disponib; /* contoare de memorie disponibilă */
static Object *obiecte; /* adresa primului obiect */
static Object *first, *last; /* primul respectiv ultimul obiect din listă */
static int nobj; /* numărul descriptorilor obiect */
```

Variabila *mdisp* de tip *long* conţine numărul de octeţi disponibili din zona de date obiect (incluzînd orice zonă de memorie neutilizată). Varibila *disponib* păstrează numărul de octeţi disponibili după ultima zonă alocată. Pointerul *obiecte* referă primul obiect din şirul descriptorilor obiect, iar pointerii *first* şi *last* referă primul, respectiv ultimul descriptor din listă.

Variabila *nobj* contorizează numărul de obiecte.

6.2.2. Rutinele alocatorului

Tabelul 6.1 conţine rutinele alocatorului de obiecte ce gestionează memoria heap.

Rutina	Actiune
`void obj_int()`	Initializeazt alocatorul de obiecte
`Object * obj_alloc(int lung)`	Returneazt un pointer la un nou obiect cu *lung* octeti de dat

`void obj_free(Object *p)`	Elibereazt obiectul referit de *p*
`void obj_dim(Object *p, int lung)`	Redimensioneaza obiectul referit de *p* la *lung* octeti de dat
`void obj_verif()`	Verifica consistenta listei
`long obj_mdisp()`	ReturneazÑ numarul de octeti liberi din zona datelor obiect
`int obj_ddisp()`	Returneaza numarul descriptorilor obiect nealocati

Tabelul 6.1. Rutinele alocatorului de obiecte

Rutina *obj_init()* iniţializează alocatorul de obiecte.

Rutina *obj_alloc()* este asemănătoare funcţiei *malloc()*, ea returnează un pointer la tipul *object*, în locul unui pointer la tipul *char*. Similar, *obj_free()* este asemănătoare cu funcţia *free()*, dar lucrează cu un pointer de tip *object*. Funcţia *obj_dim()* este asemănătoare cu funcţia *realloc()*, ea necesitînd ca argument un pointer la un obiect rezultat dintr-un apel anterior al funcţiei *obj_alloc()* (sau *obj_dim()*). Aceste patru rutine formează nucleul alocatorului de obiecte.

Cele trei rutine rămase realizează verificările de consistenţă pentru alocator sau furnizează informaţii specifice pentru un program utilizator. Spre exemplu, rutina *obj_verif()* parcurge lista descriptorilor obiect şi verifică consistenţa ei. Rutina *obj_mdisp()* returnează numărul octeţilor rămaşi neutilizaţi în zona datelor obiect, iar *obj_disponib()* returnează numărul descriptorilor de obiect rămaşi neutilizaţi. Programele utilizator pot apela aceste rutine pentru a determina cîtă memorie heap este disponibilă.

6.2.2.1. Inițializarea alocatorului

Înainte ca programul să gestioneze obiecte, alocatorul de obiecte trebuie inițializat prin intermediul funcției *obj_init()*. Această rutină alocă memorie heap pentru zona datelor obiect și pentru șirul descriptorilor obiect. Rutina atribuie valori variabilelor de control. Codul rutinei *obj_init()* este:

```
void obj_init()
{
    char *malloc(), *calloc();

if (( baza=(unsigned char *)malloc(OBJSPAȚIU))==NULL)
    obj_err("obj_init nu poate aloca spațiu datelor obiect");
if((obiecte=(Object *)calloc(OBJDIM,sizeof(Obiect)))==NULL)
    obj_err("obj_init nu poate aloca spațiu descriptorilor obiect");
obiecte[0].pobj=baza;
first = last = &obiecte[0];
nobj = 1; disponib = mdisp = OBJSPAȚIU;
}
```

Rutina apelează funcția *malloc()* pentru alocarea de memorie heap pentru zona de date obiect și funcția *calloc()* pentru alocarea de memorie heap pentru descriptorii obiect. Este singura situație cînd alocatorul utilizează rutinele standard de alocare. Variabila de control *baza* este inițializată cu adresa de început a zonei datelor obiect, iar variabila *obiecte* referă șirul de descriptori obiect.
Alocatorul de obiecte nu trebuie să elibereze zonele alocate decît înaintea terminării programului utilizator.

Rutina *obj_init()* apelează funcția *calloc()* pentru a inițializa cu zero toate cîmpurile șirului descriptorilor obiect. Acest lucru este important la alocarea de noi obiecte.
Rutina inițializează *obiecte[0]* (primul descriptor obiect) la adresa de început a zonei datelor obiect. Acesta este un obiect redundant, deoarece nu se referă la date reale, ci servește doar ca referință.
Restul rutinei *obj_init()* inițializează variabile de control. Inițial *disponib* și *mdisp* conțin întreaga zonă de date obiect, iar *nobj* este inițializat cu *1*. Pointerii *first* și *last* referă primul descriptor obiect și formează scheletul listei înlănțuite de descriptori obiect.

Figura 6.2. ilustrează memoria heap după inițializare.

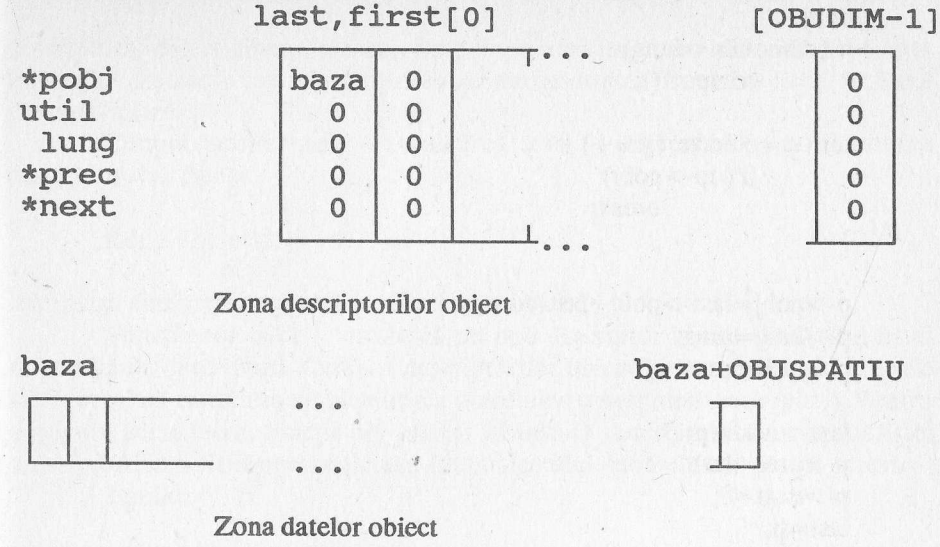


Figura 6.2. Memoria heap inaintea inițializării alocatorului

Un obiect este marcat printr-un număr între paranteze drepte ([0],[1] etc). Cîmpurile structurii fiecărui obiect sînt inițializate cu zero mai puțin cîmpul *pobj* al obiectului de referință. *Object[0]* este obiectul de referință (șablon), iar pointerul *pobj* conține adresa zonei datelor obiect.
La apariția unei erori, funcția *obj_init()* apelează funcția *obj_err()*. Acest lucru constrînge alocatorul de obiecte să termine programul, după afișarea unui mesaj de eroare în fișierul standard de erori.

6.2.2.2. Alocarea obiectelor

Funcția *obj_alloc()* permite programelor utilizator să aloce în memoria heap spațiu pentru noi obiecte. Funcția necesită ca argument numărul de octeți necesari datei obiectului. Codul funcției *obj_alloc()* este:

```
Object *
obj_alloc( int lung)
{
        register Object *p;
        if ( lung > mdisp || nobj >= OBJDIM)
                obj_err("obj_alloc: depășire spațiul datelor obiect");
```

```
        nobj++;
        if ( disponib < lung)
                compact();

        for ( p=obiecte; ; p++) {
                if ( !p->pobj)
                        break;
        }

        p->pobj=last->pobj+last->lung;
        p->lung=lung;
        p->util=0;

        last->next=p;
        p->prec=last;
        p->next=0;
        last=p;

        mdisp -= lung;
        disponib -= lung;
        DBUG(12,"obj_alloc: pobj=%x pentru %d octeti
                mdisp=%x\n", p->pobj, lung, mdisp);
        return p;
}
```

Înainte de alocarea propriu-zisă a unui nou obiect, funcţia *obj_alloc()* face cîteva verificări. Mai întîi, funcţia se asigură că există spaţiu suficient pentru *lung* octeţi în zona datelor obiect. Variabila *mdisp* conţine numărul de octeţi disponibili, incluzînd locurile rezultate din eliberarea obiectelor anterior alocate. Ca o a doua verificare, funcţia *obj_alloc()* se asigură că numărul cîmpurilor ocupate în şirul descriptorilor (*nobj*) este mai mic decît valoarea maximă *OBJDIM*. Dacă una dintre condiţii nu este îndeplinită, atunci *obj_alloc()* apelează funcţia *obj_err()* cu un mesaj de eroare. În caz contrar, *obj_alloc()* incrementează *nobj* pentru actualizarea numărului descriptorilor obiect.

În continuare, funcţia verifică memoria disponibilă la sfîrşitul zonei datelor obiect. Dacă variabila *disponibil* este mai mică decît variabila *lung*, funcţia *obj_alloc()* compactează, cu ajutorul funcţiei *compact()*, zona datelor obiect, pentru a face loc noului obiect. Funcţia *compact()* o vom descrie mai tîrziu.

Funcţia *obj_alloc()* găseşte un descriptor obiect neutilizat printr-o căutare liniară în şirul descriptorilor, după prima valoare *pobj* ce are valoarea *0*. O altă modalitate este de-a gestiona o listă a descriptorilor liberi. Astfel, toţi

descriptorii obiect liberi folosesc cîmpul pointer *next* pentru a forma o listă înlănţuită. În acest caz, alocatorul alege un descriptor liber din listă, în locul căutării liniare.

Cîmpul pointer *pobj* are valoarea zero la iniţializare şi la eliberarea descriptorului. Bucla:

```
for( p=obiecte; ;p++)
```

realizează parcurgerea şirului descriptorilor obiect.

Funcţia *obj_alloc()* instalează un nou descriptor obiect la sfîrşitul listei înlănţuite de descriptori. Ordinea descriptorilor obiect în lista înlănţuită trebuie să fie corelată cu adresa de memorie a obiectelor corespunzătoare (*pobj*). Pentru a asigura acest lucru, funcţia *obj_alloc()* instalează descriptorul la sfîrşitul listei descriptorilor şi alocă spaţiul pentru data obiectului după ultimul obiect alocat.

Liniile:

```
p->obj=last->pobj+last->lung;
p->lung=lung;
```

atribuie o adresă heap noului obiect alocat după ultimul obiect şi actualizează dimensiunea obiectului la *lung* octeţi. Funcţia *obj_alloc()* actualizează apoi cu zero cîmpul *util*.

Acest cîmp permite programelor utilizator realizarea de mici ajustări în memoria "proprie", fără a determina mutarea datelor.

Instalarea noului descriptor la sfîrşitul listei înlănţuite se realizează prin:

```
last->next=p;
p->prec=last;
p->next=o;last=p;
```

După ajustarea variabilelor de control *mdisp* şi *disponibil* cu *lung* octeţi, funcţia returnează un pointer la obiectul alocat.

Se consideră un program utilizator ce alocă două obiecte de *50*, respectiv *75* de octeţi. Figura 6.3. ilustrează memoria heap după cele două apeluri ale funcţiei *obj_alloc()*.

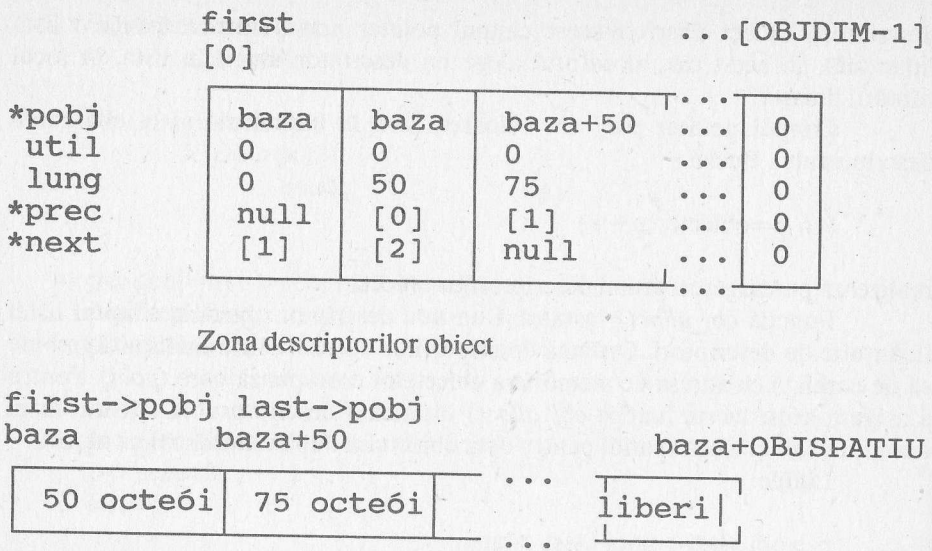


Figura 6.3. Alocarea de memorie pentru obj_alloc(50) şi obj_alloc(75)

În lista descriptorilor, *obiecte[1]* şi *obiecte[2]* conţin 50 respectiv 75 octeţi de date. Datele pentru *obiecte[1]* se găsesc la adresa *baza*, iar datele pentru *obiecte[2]* se găsesc la adresa *baza+50*. Alocatorul accesează şi controlează descriptorii obiect prin pointerii de legătură. *Obiecte[0]* (primul obiect) nu conţine date în zona datelor obiect (cîmpul *lung* este zero), iar cîmpul *next* este legătura lui spre obiectul *obiecte[1]*. Analog, cîmpul *prec* al obiectului *obiecte[2]* este legătura lui spre obiectul *obiecte[1]*. Cîmpul *next* al obiectului *obiecte[2]* este zero, astfel încît el marchează sfîrşitul listei descriptorilor obiect.

În zona datelor obiect, adresa *baza+125* este următoarea adresă de memorie disponibilă. Variabila *disponib* este iniţializată cu numărul de octeţi liberi ilustraţi în figura 6.3. Alocatorul utilizează variabilele *first* şi *last* pentru a accesa date unui obiect şi pentru a găsi următoarea adresa diponibilă.

6.2.2.3. Eliberarea obiectelor

Pentru a elibera un obiect, programele utilizator vor apela funcţia *obj_free()*,avînd ca parametru pointerul la obiectul ce se eliberează. Funcţia marchează obiectul ca fiind liber şi îl elimină din lista descriptorilor. Funcţia

semnalează eroare dacă un program utilizator încearcă să elibereze un obiect eliberat deja. Dacă un obiect este primul sau ultimul în lista descriptorilor, funcţia actualizează variabilele de control şi actualizează pointerii. Codul funcţiei *obj_free()* este:

```
void obj_free( Obiect *p)
{
        if ( p->pobj == 0)
                obj_err("obj_free: obiect deja liber");

        if ( p==last) {
                disponib += p->lung;
                last = p->prec;
                last->next=0;
        } else
        if ( p == first) {
                first = p->next;
                first->prec=0;
        } else {
                p->prec->next=p->next;
                p->next->prec=p->prec;
        }

        mdisp += p->lung;
        DBUG( 12,"obj_free: pobj=%x pentru %4d octeţi
                mdisp=%x\n",p->pobj, p->lung, mdisp);

        p->pobj = 0;
        nobj--;
}
```

Un obiect este considerat liber cînd cîmpul *pobj* are valoarea *0*. Funcţia testează în liniile următoare dacă obiectul este ultimul sau primul din listă. Se actualizează pointerii *first* şi *last* la următorul, respectiv la precedentul obiect dacă obiectul este primul respectiv ultimul din listă. Dacă, însă, obiectul nu este nici primul şi nici ultimul în listă, instrucţiunile:

```
        p->prec->next = p->next;
        p->next->prec = p->prec;
```

elimină obiectul din lista descriptorilor. Funcția *obj_free()* actualizează variabila *disponib* doar dacă obiectul este ultimul din listă. Întotdeauna *obj_free()* actualizează variabila *mdisp* și decrementează variabila *nobj*.

Dacă se eliberează cei *50* de octeți corespunzători obiectului *obiecte[1]*, se actualizează lista înlănțuită, pentru a referi peste obiectul *obiecte[1]*. Deoarece acest descriptor nu mai aparține listei înlănțuite, valorile din cîmpurile *lung*, *util*, *prec* și *next* nu mai sînt folositoare.

6.2.3. Redimensionarea obiectelor

Schimbarea dimensiunii datelor unui obiect este de regulă o operație anevoioasă. Dacă alocatorul necesită creșterea dimensiunii datelor peste limita datelor obiectului următor, este necesară mutarea datelor obiectului.

Programele utilizator schimbă dimensiunea unei date obiect cu ajutorul funcției *obj_dim()*. Funcția necesită două argumente: adresa unui descriptor obiect și o nouă dimensiune (*lung*). Apelul funcției *obj_dim()* se folosește, de regulă, pentru creștera dimensiunii obiectului, dar este utilizabil și pentru micșorarea dimensiunii sale. În consecință, programele utilizator pot modifica cîmpul *util* pentru a micșora dimensiunea unui obiect. Modificarea cîmpului *util* nu determină alocatorul de obiecte să elibereze memorie. Codul funcției *obj_dim()* este :

```
void obj_dim( Object *p, int lung)
{
        register Object *t;
        unsigned char *tmp;
        Object *new;
        int nece, disp;
        if (p->pobj==0)
                obj_err("obj_dim: necesar %d (ocupat %d)\n",
                        lung,p->lung);
                if ( p->lung >= lung) {
                        if (p==last)
                                disponib += p->lung -lung;
                        mdisp += p->lung - lung;
                        p->lung = lung;
                        DBUG( 12, "obj_dim: pobj=%x pentru %d
                         octeți mdisp=%x\n",p->pobj, lung, mdisp);
                        return;
                }
```



```
                nece = lung - p->lung;
                if ( nece > mdisp)
                 obj_err("obj_dim:depășirea spațiului datelor obiect");
                disp = 0;
                if ( p!=last || nece>disponib) {
                        for ( t=p, disp=0; ; t=t->next) {
                                if (t==last) {
                                        compact();
                                        break;
                                }
                                disp += t->next->pobj -
                                        (t->pobj+t->lung);
                                if (disp >= nece)
                                        break;
                        }
                        while (t!=p) {
                                if (t==last) {
                                        tmp=t->pobj+t->lung-1+nece;
                                        disponib -= nece;
                                }
                                else
                                        tmp = t->next->pobj-1;
                                drmove( tmp, t->pobj+t->lung-1,
                                        t->lung);
                                tmp -= t->lung;
                                t->pobj = tmp+1;
                                t=t->prec;
                        }
                }
                else
                        disponib -= lung - p->lung;
                mdisp -= lung - p->lung;
                p->lung = lung;
                DBUG( 12,"obj_dim: pobj=%x pentru %4d octeți
                        mdisp=%x\n",p->pobj, lung, mdisp);
}
```

Funcția *obj_dim()* apelează funcția *obj_err()* dacă un program utilizator încearcă să redimensioneze un obiect liber. Dacă dimensiunea cerută (*lung*) este mai mică decît dimensiunea spațiului alocat obiectului (*p->lung*), alocatorul nu trebuie să compacteze datele sau să verifice limita (frontiera) obiectelor. În acest

caz, funcția *obj_dim()* ajustează cîmpul *lung* al obiectului cu noua valoare și actualizează variabila *mdisp* și, eventual, variabila *disponib*.

Dacă dimensiunea cerută *(lung)* este mai mare decît dimensiunea spațiului alocat obiectului, funcția *obj_dim()* verifică dacă există spațiu adițional de date obiect disponibil. Dacă obiectul este ultimul și există suficient spațiu în *disponib* alocatorul nu trebuie să mute datele. Acesta este cazul cel mai simplu. Se poate însă întîmpla ca, existînd suficientă memorie liberă pentru redimensionare, să fie totuși necesară mutarea obiectelor întrucît spațiul este fragmentat. În acest caz funcția *obj_dim()* realizează mutarea datelor. Se începe cu obiectul care necesită redimensionare și se parcurge lista descriptorilor obiect pînă la întîlnirea ultimului obiect (*last*), sau pînă la găsirea spațiul necesar, rezultat din însumarea spațiilor libere dintre datele obiect. Dacă se ajunge la ultimul descriptor obiect, funcția compacteză toate datele obiect din memorie. Pentru a se crea acest loc, datele obiect sînt deplasate, octet cu octet, spre dreapta (de la primul spre ultimul obiect). Dacă *obj_dim()* găsește suficient spațiu liber înainte de-a ajunge la ultimul obiect, funcția se oprește și sînt deplasate spre dreapta doar datele obiect ce includ spațiul liber necesar.

Funcția *obj_dim()* realizează acestea prin două bucle. Prima buclă (*for*) parcurge lista descriptorilor, începînd de la pointerul *p*.

```
for ( t=p, disp=0; ; t=t->next)
```

Variabila locală *disp* acumulează spațiul liber (nealocat) existent între datele obiect:

```
disp += t->next->pobj - (t->pobj + t->lung);
```

Bucla se termină cînd este acumulat suficient spațiu:

```
disp >= nece
```

sau cînd este regăsit ultimul descriptor obiect:

```
t == last
```

A doua buclă (*while*) pornește de unde s-a încheiat prima buclă (*for*). Se parcurge înapoi lista descriptorilor obiect, mutînd datele obiect succesiv spre dreapta cu cantitatea necesară (*nece*).
Funcția *obj_dim()* apelează funcția *drmove()* după cum urmează:

```
drmove( tmp, t->pobj + t->lung - 1, t->lung);
```


Primul argument al funcției este adresa destinație, al doilea este adresa sursă, iar ultimul este numărul de octeți care se deplasează. Variabilele *tmp* și *t->pobj+t->lung-1* sînt adresele de sfîrșit pentru destinație, respectiv sursă. Mutarea datelor unui obiect începe cu octetul de sfîrșit pentru a preveni suprascrierea datelor, în cazul în care sursă și destinație se suprapun parțial.

6.2.4. Compactarea datelor obiect

Funcția *obj_alloc()* apelează funcția *compact()* dacă spațiul de după data ultimului obiect e insuficient pentru noul obiect. Se parcurge lista descriptorilor obiect și se elimină spațiile libere dintre datele obiect. Funcția *compact()* mută datele obiect spre stînga, pentru a putea deplasa spațiul liber spre dreapta. Codul funcției *compact()* este:

```
static compact()
{
        register Obiect *p, *tmp;
        void obj_ver();

        for (p=first; p!=last; p=p->next) {
                tmp=p->next;
                if (p->pobj+p->lung < tmp->pobj) {
                 stmove(p->pobj+p->lung,tmp->pobj,tmp->lung);
                 tmp->pobj = p->pobj+p->lung;
                }
        }
        if (DBUG(12, "compact: ... verificare obiect\n"))
                obj_ver();
}
```

Implementarea funcției *compact()* are la bază faptul că lista descriptorilor gestionează obiectele din memorie în ordinea crescătoare a adreselor. În timpul execuției buclei *for*, variabila *tmp* conține adresa datelor obiectului următor.

Dacă condiția:

```
( p->pobj + p->lung < tmp->pobj)
```

se evaluează la *TRUE*, există un spațiu liber de dimensiune:

```
tmp->pobj - ( p->pobj + p->lung)
```

octeți. Apelul funcției *stmove()*:

```
stmove( p->pobj + p->lung, tmp->pobj, tmp->lung);
```

mută datele (spre stînga) de la adresa *tmp->pobj* la adresa *p->pobj + p->lung*, astfel încît ele să urmeze datelor ce încep la adresa *p->pobj*. Funcția *stmove()* are același număr de argumente ca și funcția *drmove()*. Instrucțiunea:

```
tmp->pobj = p->pobj + p->lung;
```

actualizează adresa datei pentru obiectul care s-a mutat.

Se apelează funcția *obj_ver()* pentru depanare. Se va reveni la acestă funcție în secțiunea de depanare a alocatorului.

Funcțiile *drmove()* și *stmove()* necesită trei argumente: adresa destinație (*d*), adresa sursă (*s*) și numărul de de octeți ce se mută (*noct*). Codul lor este:

```
static drmove( d, s, noct)
register unsigned char *s, *d;
int noct;
{
        DBUG(12,"drmove: sursa=%x dest=%x pentru
                %4d octeți\n",s, d, noct);

        while ( noct--)
                *d-- = *s--;
}

static stmove( d, s, noct)
register unsigned char *s, *d;
int noct;
{
        DBUG(12,"stmove: sursa=%x dest=%x pentru
                %4d octeți\n", s, d, noct);

        while ( noct--)
                *d++ = *s++;
}
```


Toată acțiunea este realizată de bucla *while* și de expresiile compacte cu pointeri. Spre exemplu, în *drmove()*, codul:

```
while (noct--)
        *d-- = *s--;
```

copiază fiecare bloc de date obiect, începînd cu octeții de sfîrșit.

La scrierea funcțiilor *drmove()* și *stmove()* este necesară optimizarea, avînd în vedere apelul frecvent al acestor funcții și mutarea unor zone de date de dimensiune mare. Atîta timp cît dimensiunea unui pointer este identică cu dimensiunea unui întreg (2 octeți), utilizarea expresiilor compacte cu pointeri, cu variabile de clasă *register* pentru *s* și *d*, este de mare folos. Utilizarea variabilelor *register* se poate întîmpla să fie dependentă de hardul calculatorului.

De regulă, cea mai bună abordare constă în implementarea funcțiilor *drmove()* și *stmove()* cu funcția de bibliotecă C *memcpy()*. Funcția este rapidă, deoarece versiunile noi conțin cod în limbaj de asamblare ce realizează mutarea pe blocuri. Se poate întîmpla ca implementarea funcției *memcpy()* să nu permită mutarea unor blocuri ce se suprapun. Ca atare, trebuie verificat dacă implementarea funcției *memcpy()* răspunde exigențelor programului. Din acest motiv am eliminat intenționat apelul *memcpy()*, deoarece codul ilustrat pentru funcțiile *drmove()* și *stmove()* este mai portabil. În cazul în care calculatorul permite apel *memcpy()* cu suprapunerea șirului destinație cu șirul sursă, se înlocuiește apelul *drmove()* și *stmove()* cu *memcpy()*.

6.2.5. Verificarea consistenței

Capitolul 5 a ilustrat cîteva modalități de depanare a unor programe mai complexe. Alocatorul de obiecte este un exemplu bun de program ce poate conține defecte ascunse. Alocarea și eliberarea pointerilor poate fi supărătoare odată ce simptomele pot să nu se repete.

Pentru a preveni această situație, implementăm funcția *obj_ver()*, care realizează veficarea consistenței zonei datelor și descriptorilor obiect. Verificarea consistenței semnifică testarea corectitudinii unor relații cunoscute ce există între anumite variabile și semnalarea eventualelor discrepanțe. Programele utilizator pot apela această rutină în orice moment în care se dorește testarea integrității alocatorului. Spre exemplu, alocatorul de obiecte utilizează valoarea *0* în cîmpul *pobj* din lista descriptorilor pentru a marca că acel obiect este neutilizat. Ca atare, numărul descriptorilor obiect ce au valoarea acestui cîmp diferită de *0* trebuie să fie egal cu numărul obiectelor alocate; sau suma dimensiunii memoriei alocate și nealocate pentru datele obiect trebuie să fie egală cu dimensiunea totală a zonei

datelor obiect. Bineînțeles, fără ca data unui obiect să se suprapună peste data altui obiect.

Funcția *obj_ver()* realizează această verificare de consistență. Deoarece funcția consumă timp programele utilizator este bine s-o apeleze doar în timpul implementării și depanării lor. Se recomandă eliminarea apelului ei din cod, prin compilare condiționată. Iată codul funcției *obj_ver()*:

```
void obj_ver()
{
        Object *p;
        register int i, noct, lung;

        for ( noct=lung=i=0; i<OBJDIM;i++) {
          if (obiecte[i].pobj !=0) {
              noct++;
              lung += obiecte[i].lung;
          }
        }
        if (lung + mdisp != OBJSPAȚIU)
          obj_err("eroare obj_ver in spațiul datelor obiect");
        if ( noct != nobj)
          obj_err("eroare obj_ver pentru descriptorii obiect");

        for (i=1, p=first; ; i++, p=p->next) {
          if ( p==last || i>nobj)
              break;
          if (p->pobj +p->lung > p->next->pobj)
              obj_err("eroare obj_ver; depasire spațiu datelor obiect");
        }
        if (p!=last || i !=nobj)
          obj_err("eroare la verificarea consistenței");
}
```

Funcția *obj_ver()* utilizează două bucle *for* pentru a parcurge lista descriptorilor obiect. Prima buclă traversează șirul descriptorilor (pe baza indexului) și contorizează numărul de obiecte și numărul de octeți alocați datelor obiect. Instrucțiunea:

```
        if (lung + mdisp != OBJDIM)
```


testează dacă suma dimensiunilor zonei alocate și nealocate din memorie este egală cu dimensiunea zonei datelor obiect (*OBJDIM*). Similar, instrucțiunea:

```
        if ( noct != nobj)
```

testează dacă numărul obiectelor alocate (*noct*) coincide cu numărul curent al descriptorilor obiect (*nobj*). Dacă unul dintre teste eșuază, funcția *obj_ver()* afișează un mesaj de eroare și programul se termină.

A doua buclă *for* parcurge lista descriptorilor prin cîmpul de legătură (*next*). Parcurgerea se oprește cînd se depășește numărul de obiecte alocate (*nobj*), sau cînd am regăsit valoarea ultimului descriptor din listă (*last*).

Pentru fiecare descriptor, funcția *obj_ver()* verifică suprapunerea zonelor de memorie prin instrucțiunea:

```
        if (p->obj + p->lung > p->next->pobj)
```

Ultimul test verifică dacă bucla s-a terminat corespunzător. Acest lucru înseamnă că trebuie să fi ajuns la ultimul obiect în lista descriptorilor obiect și numărul obiectelor alocate trebuie să fie egal cu *nobj*. Dacă cel puțin o condiție nu este adevărată, se apelează funcția *obj_err()* și programul se termină. Singurele rutine neanalizate ale alocatorului au rămas *obj_mdisp()*, *obj_ddisp()* și *obj_err()*. Iată codul lor:

```
long obj_mdisp()
{
        return mdisp;
}

int obj_ddisp()
{
        return OBJDIM-nobj;
}

static obj_err( char *msg)
{
        fprintf(stderr, "objalloc: %s\n" ,msg);
        exit(1);
}
```

Funcția *obj_mdisp()* returnează variabila de control *mdisp*. Aceasta permite programelor utilizator să cunoască cît din spațiul datelor obiect a rămas

neutilizat. Asemănător, *obj_ddisp()* returnează programelor utilizator numărul descriptorilor obiect neutilizați.

Funcția *obj_err()* este declarată *static* și nu este apelabilă din programele utilizator. La apariția unei erori, funcția *obj_err()* afișează mesajele de eroare în fișierul standard de erori și termină programul cu un apel *exit(1)*.

6.2.6. Interfața alocatorului

Pentru a testa alocatorul de obiecte, se scrie un program de interfață. Programul de interfață conține funcția principală *main()*, care apelează funcțiile alocatorului și simulează un program utilizator. Programul de interfață este compilat cu opțiunea *-D*, pentru a testa funcțiile alocatorului. Fără această opțiune, compilatorul creează un fișier relocabil ce poate fi executat cu alte programe.

Programul actual de interfață conține un program principal și o funcție *probiecte()*. Programul principal apelează funcția *obj_init()* pentru a inițializa alocatorul și funcția *obj_alloc()* pentru a aloca memorie pentru cinci obiecte. Programul folosește un șir de pointeri la obiecte pentru a gestiona obiectele. Programul de interfață compactează, eliberează și redimensionează obiectele și apelează funcția *obj_ver()* pentru verificarea consistenței obiectelor. Funcția *probiecte()* afișează lista înlănțuită a descriptorilor obiect, afișînd adresa fiecărui obiect, dimensiunea sa și spațiul neutilizat.

Iată codul pentru programul de interfață:

```
#ifdef DRIVER

main()
{
        Object *p[5];
        int i;
        void obj_init(), obj_free(),
                obj_dim(), obj_ver();
        obj_init();
        obj_ver();
        p[0]=obj_alloc( 100);
        p[1]=obj_alloc( 200);
        p[2]=obj_alloc( 300);
        p[3]=obj_alloc( 400);
        p[4]=obj_alloc( 500);
        probiecte( stderr);
```



```
        obj_free( p[2]);
        obj_ver();
        obj_dim(p[0], 64);
        obj_dim(p[3], 275);
        probiecte( stderr);
        obj_ver();
        obj_dim(p[1], 800);
        obj_ver();
        probiecte(stderr);
}

probiecte( FILE *unde)
{
    Object *p;
    long obj_mdisp();
    int nobj;
    fprintf( unde,"Obiect\t\tAdresa\t\tDimensiune\tLiber\n");
    for (nobj=0, p=first; p!=last; nobj++, p=p->next)
        fprintf(unde,"%4d\t\t%4d\t\t%4d\t\t%4d\n", nobj,
            p->pobj, p->lung, p->next->pobj-(p->pobj+p->lung));
    fprintf(unde,"%4d\t\t%4d\t\t%4d\n",nobj,last->pobj, last->lung);
    fprintf(unde,"Spațiul rămas în zona datelor obiect:%7ld octeți\n",
            obj_mdisp());
    fprintf(unde,"Spațiul rămas în zona descriptorilor obiect: %6d\n",
            obj_ddisp());
}

#endif
```

Programul principal alocă cinci obiecte de dimensiuni diferite. Programul eliberează al treilea obiect (*p[2]*) și apelează *obj_ver()* pentru verificarea consistenței. Funcția *main()* apelează apoi de trei ori funcția *obj_dim()* pentru a redimensiona obiectele. Primele două apeluri reduc dimensiunea datelor obiect, iar al treilea mărește de opt ori dimensiunea primului obiect. Se apelează funcția *probiecte()* pentru afișarea de informații despre obiecte.

Funcția *probiecte()* afișează mesajele în fișierul standard de erori, dar la fel de bine poate să le afișeze într-un fișier de date. Se parcurge lista descriptorilor obiect afișîndu-se aderesa fiecărui obiect, dimensiunea sa și spațiul liber. Prin expresia:

p->next->pobj - (p->pobj + p->lung)

funcţia determină numărul de octeţi liberi existenţi între obiectul curent şi cel următor. Funcţia apelează *obj_mdisp()* pentru afişarea spaţiului liber din zona datelor obiect şi *obj_ddisp()* pentru afişarea numărul de obiecte folosite.

Tot ce a rămas de făcut este compilarea şi execuţia funcţiilor alocatorului, împreună cu depanatorul din fişierul **dbug.c**, prezentat deja în capitolul anterior.

tcc -I:\tc -L:\tc -DDRIVER objalloc.c dbug.o
objalloc

Obiect	*Adresa*	*Dimensiune*	*Liber*
0	*2774*	*0*	*0*
1	*2774*	*100*	*0*
2	*2874*	*200*	*0*
3	*3074*	*300*	*0*
4	*3374*	*400*	*0*
5	*3774*	*500*	

Spaţiul rămas în zona datelor obiect:27172 octeţi
Spaţiul rămas în zona descriptorilor obiect: 506

Obiect	*Adresa*	*Dimensiune*	*Liber*
0	*2774*	*0*	*0*
1	*2774*	*6436*	
2	*2874*	*200*	*300*
3	*3374*	*275*	*125*
4	*3774*	*500*	

Spaţiul rămas în zona datelor obiect:27633 octeţi
Spaţiul rămas în zona descriptorilor obiect: 507

Obiect	*Adresa*	*Dimensiune*	*Liber*
0	*2774*	*0*	*0*
1	*2774*	*64* *0*	
2	*2838*	*800*	*0*
3	*3638*	*275*	*0*
4	*3913*	*500*	

Spaţiul rămas în zona datelor obiect:27033 octeţi
Spaţiul rămas în zona descriptorilor obiect: 507

Programul apelează *probiecte()* de trei ori, aşa încît rezultatul execuţiei apare în trei secţiuni. Prima secţiune prezintă lista descriptorilor după alocarea memoriei

pentru cele cinci obiecte. Adresele memoriei heap sînt în hexazecimal şi coloana *Liber* arată că nu există locuri libere. În ultima poziţie a coloanei *Liber* nu apare nimic, deoarece restul spaţiului datelor obiect este disponibil. Există *506* descriptori obiect rămaşi.

A doua secţiune prezintă lista descriptorilor după eliberarea obiectului al treilea şi redimensionarea primului şi celui de-al patrulea obiect. Coloana *Liber* conţine numărul de octeţi rămaşi neutilizaţi din fiecare obiect. Există *507* descriptori obiect liberi.

Ultima secţiune prezintă lista descriptorilor după redimensionarea obiectului *p[1]*. Întrucît nu există suficient spaţiu pentru realizarea expandării, alocatorul compactează zona datelor obiect şi rearanjează datele. Coloana *Liber* arată că nu există spaţiu liber între obiecte. Numărul descriptorilor obiect nu se schimbă şi necesarul de spaţiu disponibil din zona datelor obiect descreşte.

Programul de interfaţă este util în studiul comportării alocatorului. Ca exerciţiu, apelaţi funcţiile de alocare cu diverşi parametri şi analizaţi rezultatul execuţiei.

6.2.7. Depanarea alocatorului

Fiecare funcţie a alocatorului conţine înglobat în codul ei apeluri de funcţii de depanare. Utilizînd tehnica prezentată în capitolul 5 putem afişa mesaje de depanare în timpul execuţiei programului. De regulă, apelul funcţiei de depanare are formatul:

```
DBUG(12,"obj_alloc: pobj=%x pentru %4d octeţi mdisp =
        %x\n", p->pobj, lung, mdisp);
```

DBUG este un macrou definit în **dbug.h**. Acesta apelează funcţia de depanare dacă numărul de fanion *12* apare în fişierul **flags**, din directorul curent. Dacă acesta nu există, execuţia se face fără afişarea mesajelor de depanare. Această tehnică permite vizualizarea informaţiilor de depanare, fără recompilarea modulelor program.

Iată rezultatul execuţiei pentru programul de interfaţă:

obj_alloc:pobj=ffb8 pentru 100 octeţi mdisp=7ae8
obj_alloc:pobj=ffb8 pentru 200 octeţi mdisp=7af2
obj_alloc:pobj=ffb8 pentru 300 octeţi mdisp=7afc
obj_alloc:pobj=ffb8 pentru 400 octeţi mdisp=7b06
obj_alloc:pobj=ffb8 pentru 500 octeţi mdisp=7b10

```
Obiect          Adresa          Dimensiune      Liber
0               2774             0              0
1               2774            100             0
2               2874            200             0
3               3074            300             0
4               3374            400             0
5               3774            500
Spațiul rămas în zona datelor obiect:27172 octeți
Spațiul rămas în zona descriptorilor obiect: 506
obj_free:pobj=ffba pentru 2743 octeți mdisp=7afc
obj_dim:pobj=ffb0 pentru 31464 octeți mdisp=82
obj_dim: pobj=ffb0 pentru 31494 octeți mdisp=82
Obiect          Adresa          Dimensiune      Liber
0               2774             0              0
1               2774             6436
2               2874            200             300
3               3374            275             125
4               3774            500
Spațiul rămas în zona datelor obiect:27633 octeți
Spațiul rămas în zona descriptorilor obiect: 507
stmove:sursa=ff9c dest=b3a pentru 2838 octeți
stmove: sursa=ff9c dest=d2e pentru 3038 octeți
stmove: sursa=ff9c dest=ebe pentru 3313 octeți
compact: ... verificare obiect
drmove: sursa=ffa4 dest=ee4 pentru 4412 octeți
drmove: sursa=ffa4 dest=cf0 pentru 3912 octeți
obj_dim:pobj=ffb0 pentru 31474octeți mdisp=7af2
Obiect          Adresa          Dimensiune      Liber
0               2774             0              0
1               2774             640
2               2838            800             0
3               3638            275             0
4               3913            500
Spațiul rămas în zona datelor obiect:27033 octeți
Spațiul rămas în zona descriptorilor obiect: 507
```

Afișarea conține toate părțile alocatorului. Aceasta include compactarea, mutarea, redimensionarea, alocarea și eliberarea obiectelor. Înterpretarea rămîne la latitudinea cititorului.

A1. Anexă - Funcții video Turbo C 2.0

Anexa de față prezintă modul de definire și de utilizare a funcțiilor video Turbo C 2.0. Ea constituie o completare necesară (și promisă) a **Ghidului de utilizare Turbo C 2.0.**

Orice adaptor video poate lucra în cîteva moduri diferite. Modul specifică numărul de coloane pe ecran (mod text), rezoluția (mod grafic) și tipul de afișare (color sau alb/negru).

A1.1. Modul text

Ecranul este împărțit în 40 sau 80 de coloane a cîte 25 de linii. Fiecare 'celulă' constă din 2 părți, avînd corespondent cîte un octet în memoria ecran:

- caracterul de afișat (codul ASCII);
- atributul de afișare (culoare, intensitate).

O zonă dreptunghiulară de ecran se numește **fereastră**. Ieșirea pe ecran se face în fereastra activă, restul ecranului rămînînd nemodificat. Implicit, fereastra activă este ecranul întreg, dar dimensiunea și poziția ei pot fi modificate de utilizator.

În mod text, colțul stînga sus are coordonatele (1,1), ele crescînd în jos și spre dreapta. Dacă se lucrează cu ferestre, coordonatele se dau relativ la acestea, și nu față de ecranul întreg. Funcțiile pentru acest mod lucrează în toate cele 5 moduri video text posibile.

În Turbo C, funcțiile directe de intrare/ieșire pentru consolă sînt extrem de evoluate. Ele fac parte din biblioteca standard, prototipurile lor fiind definite în fișierul **conio.h**. Ca urmare, orice program care utilizează aceste funcții trebuie să conțină linia:

```
#include <conio.h>
```

A1.1.1. Funcții pentru afișarea și tratarea textului

int cprintf(const char (format [,argument, ...]);

Afișează pe ecran informația formatată. Argumentele sale sînt identice cu ale lui **printf**, dar informația este scrisă ținîndu-se cont de culorile

de text și fond stabilite. De asemenea, caracterul \n nu este transformat automat în perechea \n\r. Funcția returnează numărul de caractere scrise.

int cputs(const char *sir);

Afișează pe ecran șirul *sir*, fără a adăuga un caracter de linie nouă. Caracterul \n nu este transformat automat în perechea \n\r. Funcția returnează ultimul caracter scris.

int putch(int c);

Afișează caracterul *c* în fereastra de text curentă. Caracterul \n nu este transformat automat în perechea \n\r. Funcția returnează caracterul scris în caz de succes, respectiv EOF în caz de eroare.

int getche(void);

Citește un caracter unic de la consolă și îl afișează în fereastra de text curentă. Funcția returnează caracterul citit.

void clrscr(void);

Șterge fereastra de text curentă și poziționează cursorul în punctul de coordonate (1, 1).

void clreol(void);

Șterge toate caracterele de la poziția curentă a cursorului pînă la sfîrșitul liniei, fără a deplasa cursorul.

void delline(void);

Șterge linia în care se află poziționat cursorul și mută toate liniile de dedesubt cu o linie în sus.

void gotoxy(int x, int y);

Mută cursorul în punctul de coordonate *(x, y)* din fereastra de text curentă. Dacă se specifică coordonate invalide, apelul funcției este ignorat.

void insline(void);

Inserează o linie goală în fereastra de text curentă la poziția cursorului, utilizînd culoarea de fond curentă. Liniile de desubt sînt mutate în jos cu o linie, iar ultima linie din fereastră dispare.

int movetext(int stinga, int sus, int dreapta, int jos, int st_nou, int sus_nou);

Copiază conținutul zonei dreptunghiulare de ecran definită de *stinga, sus, dreapta, jos* într-o nouă zonă de aceeași dimensiune, avînd coordonatele colțului stînga sus definite de *(st_nou, sus_nou)* (relativ la ecranul întreg). Funcția returnează o valoare nenulă la succes, respectiv 0 în caz de eșec.

int gettext(int stinga, int sus, int dreapta, int jos, void *destin);

Copiază conținutul zonei dreptunghiulare de ecran definită de *stinga, sus, dreapta, jos* într-o zonă de memorie pointată de *destin*. Coordonatele se dau relativ la ecranul întreg. Fiecare poziție ecran ocupă doi octeți în memorie: primul reprezintă codul caracterului, al doilea atributele video ale celulei citite. Funcția returnează 1 în caz de reușită, respectiv 0 în caz de eșec.

int puttext(int stinga, int sus, int dreapta, int jos, void *sursa);

Copiază conținutul zonei de memorie pointate de *sursa* într-o zonă dreptunghiulară de ecran definită de *stinga, sus, dreapta, jos*. Coordonatele se dau relativ la ecranul întreg. Funcția returnează o valoare nenulă în caz de reușită, respectiv 0 în caz de eșec.

A1.1.2. Controlul ferestrelor și a modului video

void window((int stinga, int sus, int dreapta, int jos);

Definește pe ecran o fereastră de text, de cordonate *stinga, sus, dreapta, jos*. Dimensiunea minimă este 1*1, iar cea implicită ecranul întreg: (1, 1, 80, 25) pentru mod 80 coloane, respectiv (1, 1, 40, 25) pentru mod 40 coloane. Dacă se specifică coordonate invalide, apelul funcției este ignorat.

void textmode(int mod_nou);

Selectează un nou mod text. La apelul său, fereastra curentă este poziționată la ecranul întreg, iar atributele video la valorile normale. Argumentul *mod_nou* poate fi specificat prin următoarele constanta simbolice, definite în **conio.h**:

Constantă simbolică	Valoare numerică	Mod text
LASTMODE	-1	Modul text anterior
BW40	0	Alb/negru, 40 coloane
C40	1	Color, 40 coloane
BW80	2	Alb/negru, 80 coloane
C80	3	Color, 80 coloane
MONO	7	Monocrom, 80 coloane

A1.1.3. Controlul atributelor

void textcolor(int culoare_nouă);

Selectează culoarea de text curentă. Nu sînt afectate caracterele deja aflate pe ecran, ci doar cele afișate după apelul său, prin funcții de intrare/ieșire directe (de exemplu, **cprintf**).

void textbackground(int culoare_noua);

Selectează culoarea de fond curentă. Nu sînt afectate caracterele deja aflate pe ecran, ci doar cele afișate după apelul său, prin funcții de intrare/ieșire directe (de exemplu, **cprintf**).

void textattr(int atribut_nou);

Selectează culorile de text și fond curente. Nu sînt afectate caracterele deja aflate pe ecran, ci doar cele afișate după apelul său, prin funcții de intrare/ieșire directe (de exemplu, **cprintf**). Informația de culoare este codificată în modul următor:

Biți	Semnificație
0, 1, 2, 3	Culoare de text
4, 5, 6	Culoare de fond
8	Clipire

În fișierul **conio.h** sînt definite constante simbolice pentru culori, după cum indică tabelul de mai jos. Pentru fond sînt disponibile doar primele 8 culori, iar pentru text toate cele 16.

Constantă simbolică	Valoare numerică	Text/Fond
BLACK	0	Ambele
BLUE	1	Ambele
GREEN	2	Ambele
CYAN	3	Ambele
RED	4	Ambele
MAGENTA	5	Ambele
BROWN	6	Ambele
LIGHTGRAY	7	Ambele
DARKGRAY	8	Doar text
LIGHTBLUE	9	Doar text
LIGHTGREEN	10	Doar text
LIGHTCYAN	11	Doar text
LIGHTRED	12	Doar text
LIGHTMAGENTA	13	Doar text
YELLOW	14	Doar text
WHITE	15	Doar text
BLINK	128	Doar text

Constanta simbolică **BLINK** se poate aduna la valoarea argumentului pentru culoarea de text astfel încît acesta să clipească.

void highvideo(void);

Selectează intensitate mărită pentru caractere, poziționînd bitul de intensitate din culoarea de text. Nu sînt afectate caracterele deja aflate pe

ecran, ci doar cele afișate după apelul său, prin funcții de intrare/ieșire directe (de exemplu, **cprintf**).

void lowvideo(void);

Selectează intensitate scăzută pentru caractere, ștergînd bitul de intensitate din culoarea de text. Nu sînt afectate caracterele deja aflate pe ecran, ci doar cele afișate după apelul său, prin funcții de intrare/ieșire directe (de exemplu, **cprintf**).

void mormvideo(void);

Selectează caractere normale, atributele textului la valoarea avută la pornirea programului. Nu sînt afectate caracterele deja aflate pe ecran, ci doar cele afișate după apelul său, prin funcții de intrare/ieșire directe (de exemplu, **cprintf**).

A1.1.4. Interogarea stării

void gettextinfo(struct text_info *str);

Completează structura de tip **tex_tinfo** pointată de *str* cu informațiile video curente. Acest tip este definit în **conio.h** în modul următor:

```
struct text_info
{
        unsigned char winleft;      /* coordonata stinga a ferestrei */
        unsigned char wintop;       /* coordonata sus a ferestrei */
        unsigned char winright;     /* coordonata dreapta a ferestrei */
        unsigned char winbottom;    /* coordonata jos a ferestrei */
        unsigned char attribute;    /* atributele textului */
        unsigned char normattr;     /* atribute normale */
        unsigned char currmode;     /* BW40, BW80, C40, C80 */
        unsigned char screenheight; /* jos - sus*/
        unsigned char screenwidth;  /* dreapta - stinga*/
        unsigned char curx;         /* coordonata x in fereastra curenta */
        unsigned char cury;         /* coordonata y in fereastra curenta */
};
```


int wherex(void);

Returnează coordonata orizontală *x* a poziției curente a cursorului în fereastra de text curentă.

int wherey(void);

Returnează coordonata verticală *y* a poziției curente a cursorului în fereastra de text curentă.

A1.1.5. Variabila directvideo

Variabila **directvideo** controlează dacă ieșirea programului se va face direct în memoria video (**directvideo** = 1) sau va trece prin apeluri BIOS (**directvideo** = 0). Valoarea implicită este 1. În acest caz, se obțin cele mai bune performanțe, dar sînt necesare calculatoare 100% compatibile IBM PC. A doua variantă (0) lucrează mai lent, dar funcționează pe orice mașină compatibilă BIOS cu IBM.

A1.2. Modul grafic

Ecranul este împărțit în pixeli (elemente de imagine), fiecare afișînd un punct pe ecran. Rezoluția (numărul de pixeli) depinde de tipul de adaptor conectat și de modul video.

O zonă dreptunghiulară de ecran se numește **port de afișare**. Acesta lucrează ca un ecran virtual, restul rămînînd neschimbat. În mod grafic, colțul stînga sus are coordonatele (0,0), ele crescînd în jos și spre dreapta. Dacă se lucrează cu porturi de afișare, coordonatele se dau relativ la acestea, și nu față de ecranul întreg.

Biblioteca grafică Turbo C dispune de peste 70 de funcții grafice, de diferite niveluri, permițînd lucrul cu mai multe stiluri de desenare și seturi de caractere. Aceste funcții se află în fișierul bibliotecă **GRAPHICS.LIB**, avînd prototipul în **GRAPHICS.H**. De asemenea, există driverele pentru dispozitive grafice (fișierele ***.BGI**) și cele pentru seturi de caractere (fișierele ***.CHR**).

Lucrînd în mediul integrat, opțiunea **Options/Linker/Graphics library** trebuie poziționată pe **On**. Biblioteca grafică va fi legată automat.

Dacă se folosește utilitarul TCC, fișierul **GRAPHICS.LIB** trebuie specificat în linia de comandă. Dacă un program, numit *test*, lucrează grafic, linia de comandă va fi:

tcc test graphics.lib

Pentru toate modelele de memorie există o singură bibliotecă grafică, cu funcții de tip **far**. Întrucît funcțiile grafice lucrează cu pointeri **far**, ele nu sînt acceptate în modelul de memorie TINY. Pentru a lucra corect, toate modulele care folosesc grafică trebuie să conțină linia:

#include <graphics.h>

A1.2.1. Controlul sistemului grafic

void far closegraph(void);

Închide sistemul grafic, dealocînd memoria alocată și refăcînd ecranul în care era înainte de apelul lui **initgraph**.

void far detectgraph(int far *driver, int far *mod;)

Detectează adaptorul grafic al sistemului și alege modul care oferă cea mai bună rezoluție. Dacă nu se detectează parte de hardware pentru grafică, **driver* este poziționat pe -2, iar **graphresult** va returna tot -2.

Parametrul **driver* specifică driverul grafic de utilizat. El poate fi specificat utilizînd una din următoarele constante, definite în **graphics.h**:

Constantă simbolică	Valoare numerică
DETECT	0 (cerere autodetecție)
CGA	1
MCGA	2
EGA	3
EGA64	4
EGAMONO	5
IBM8514	6
HERCMONO	7
ATT400	8
VGA	9
PC3270	10

Parametrul **mod* specifică modul grafic inițial (dacă **driver* este diferit de **DETECT**). Valoarea sa poate fi specificată utilizînd una din următoarele constante, definite în **graphics.h**:

Driver grafic	Moduri grafice	Valoare	Coloane * Linii	Paletă	Pagini
CGA	CGAC0	0	320*200	C0	1
	CGAC1	1	320*200	C1	1
	CGAC2	2	320*200	C2	1
	CGAC3	3	320*200	C3	1
	CGAHI	4	640*200	2 culori	1
MCGA	MCGAC0	0	320*200	C0	1
	MCGAC1	1	320*200	C1	1
	MCGAC2	2	320*200	C2	1
	MCGAC3	3	320*200	C3	1
	MCGAMED	4	640*200	2 culori	1
	MCGAHI	5	640*480	2 culori	1
EGA	EGALO	0	640*200	16 culori	4
	EGAHI	1	640*350	16 culori	2
EGA64	EGA64LO	0	640*200	16 culori	1
	EGA64HI	1	640*350	4 culori	1
EGA-MONO	EGAMONOHI	3	640*350	2 culori	1*
	EGAMONOHI	3	640*350	2 culori	2**
HERC	HERCMONOHI	0	720*384	2 culori	2
ATT400	ATT400C0	0	320*200	C0	1
	ATT400C1	1	320*200	C1	1
	ATT400C2	2	320*200	C2	1
	ATT400C3	3	320*200	C3	1
	ATT400MED	4	640*200	2 culori	1
	ATT400HI	5	640*400	2 culori	1
VGA	VGALO	0	640*200	16 culori	2
	VGAMED	1	640*350	16 culori	2
	VGAHI	2	640*480	16 culori	1
PC3270	PC3270HI	0	720*350	2 culori	1
IBM8514	IBM8514HI	0	640*480	256 culori	
	IBM8514LO	0	1024*768	256 culori	

* Placă EGAMONO cu 64 Kocteți,
** * Placă EGAMONO cu 256 Kocteți

Valorile C0, C1, C2 și C3 se referă la paletele predefinite pentru CGA. Se poate selecta culoarea de fon (intrarea 0), dar celelalte culori sînt prestabilite, după cum urmează:

	Culoarea pentru valoarea de pixel		
Număr paletă	1	2	3
0	LIGHTGREEN	LIGHTRED	YELLOW
1	LIGHTCYAN	LIGHTMAGENTA	WHITE
2	GREEN	RED	BROWN
3	CYAN	MAGENTA	LIGHTGRAY

void far graphdefaults(void);

Reface toate poziționările modului grafic la valorile lor implicite.

void far _graphfreemem(void far *ptr, unsigned dim);

Biblioteca grafică apelează la această funcție pentru a elibera memoria alocată prin **_graphgetmem**. Utilizatorul poate defini o versiune proprie a acestei funcții.

void far * far _graphgetmem(unsigned dim);

Biblioteca grafică apelează la această funcție pentru a aloca memoria necesară tampoanelor interne,driverelor grafice și seturilor de caractere. Utilizatorul poate defini o versiune proprie a acestei funcții.

int far getgraphmode(void);

Returnează modul grafic curent.

void far getmoderange(int driver, int far *modlo, int far *modhi);

Returnează în *modlo*, respectiv *modhi* cel mai mic și cel mai mare mod garfic permis pentru *driver*. Dacă se spevcifică un *driver* invalid, ceilalți doi parametri vor avea valoarea -1. Dacă *driver* are valoarea -1, se returnează modurile pentru driverul curent.

void far initgraph(int far *driver, int far *mod, int far *cale);

Inițializează sistemul grafic, încărcînd driverul grafic specificat și tecînd sistemul în mod grafic. Dacă parametrul *cale este NULL, driverul grafic trebuie să fie în directorul curent. Pentru mod și driver se pot utiliza constantele simbolice definite la funcția **detectgraph**.

Funcția poziționează codul intern de eroare. În caz de succes, acesta are valoarea 0. În caz de eșec, *driver va avea una din valorile -2, -3, -4, sau -5 iar **graphresult** va returna aceeași valoare, cu semnificațiile următoare:

-2	Nu s-a detectat placă grafică
-3	Nu s-a găsit fișierul de driver
-4	Driver invalid
-5	Memorie insuficientă pentru a încărca driverul grafic

int far installuserdriver(char far *nume, int huge (*detect)(void));

Permite adăugarea unui nou driver la tabela internă .BGI. Parametrul nume reprezintă noul fișier de driver (.BGI), iar detect este un pointer la o funcție opțională de autodetecție, care nu are parametri și returnează o valoare întreagă. Valoarea returnată reprezintă numărul de driver care poate fi specificat ca parametru funcției **initgraph**.

int far installuserfont(char far *nume);

Încarcă un fișier pentru seturi de caractere (.CHR) care nu se găsește în sistemul BGI original. Valoarea returnată reprezintă numărul care poate fi specificat ca parametru funcției **settextstyle**. Dacă tabela internă pentru seturi de caractere este plină, se va returna valoarea -1.

int registerbgidriver(void (*driver)(void));

Permite încărcarea unui driver de fișier și 'înregistrarea' sa. După ce locația sa în memorie a fost transmisă funcției **registerbgidriver**, **initgraph** va utiliza acest driver. Un astfel de driver utilizator poate fi încărcat de pe disc în heap, sau poate fi convertit în fișier .OBJ și legat la fișierul .EXE. După înregistrare, rezultatul funcției **registerbgidriver** (valoarea numărului de driver) trebuie trasmis la **initgraph**. În caz de eroare, se returnează o valoare negativă.

În mod implicit, încărcarea driverelor se face prin alocare de memorie și încărcarea de pe disc a fișierului .BGI corespunzător. Această metodă dinamică poate fi înlocuită prin legarea fișierelor de drivere grafice direct în fișierul executabil. În primul rînd, fișierele .BGI trebuie transformate în fișiere .OBJ, cu utilitarul **BGIOBJ**. În fișierul sursă, înainte de apelul lui **initgraph**, se vor introduce apeluri ale funcției **registerbgidriver**, pentru a înregistra driverele grafice. La crearea programului executabil, trebuie legate fișierele .OBJ pentru drivere.

void far restorcrtmode(void);

Reface modul original detectat de **initgraph**.

unsigned far setgraphbufsize(unsigned dim);

Modifică dimensiunea tamponului grafic intern (implicit 4096 octeți). Apelul său se va face înainte de **initgraph**, fiind apoi ignorat pînă la următorul apel al lui **closegraph**. Funcția returnează dimensiunea anterioară a tamponului grafic.

void far setgraphmode(int mod);

Selectează un mod grafic diferit de cel poziționat cu **initgraph**, șterge ecranul și reface valorile implicite ale poziționărilor grafice. În cazul unui parametru invalid, funcția returnează -10.

A1.2.2. Desenare și umplere

void far arc(int x, int y, int unghi1_1, int unghi_2, int rad);

Desenează un arc de cerc circular, folosind culoarea de desen curentă, de centru *(x, y)*, între *unghi_1* și *unghi_2*, de rază *rad*. Valoarea unghiului crește în sens invers celui orar, 0 grade fiind pe ora 3, 90 de grade pe ora 12, etc.

void far circle(int x, int y, int radius);

Desenează un cerc, folosind culoarea de desen curentă, cu centrul în *(x, y)*, de rază *rad*.

void far drawpoly(int nr_virf, int far *puncte);

Desenează un poligon cu *nr_virf* virfuri, folosind culoarea de desen curentă. Parametrul **puncte* pointează la o secvență de (*nr_virf* * 2) întregi, fiecare pereche reprezentînd coordonatele *x* și *y* ale cîte unui vîrf. În caz de eroare, **graphresult** va returna valoarea -6.

void far ellipse(int x, int y, int unghi1_1, int unghi_2, int razax, int razay);

Desenează un arc eliptic, folosind culoarea de desen curentă, de centru *(x, y)*, între *unghi_1* și *unghi_2*, de rază orizontală *razax*, respectiv verticală *razay*. Valoarea unghiului crește în sens invers celui orar, 0 grade fiind pe ora 3, 90 de grade pe ora 12, etc.

void far getarccords(struct arccordstype far *arccord);

Completează o structură de tip **arccords** pointată de *arccord* cu informații referitoare la ultimul apel al funcției **arc**. Această structură este definită în **graphics.h** după cum urmează:

```
struct arccordstype
{
        int x, y;                   /* centrul */
        int xstart, ystart;         /* pozitia de inceput*/
        int xend, yend;             /* pozitia de sfirsit */
}
```

void far line(int x1, int y1, int x2, int y2);

Desenează o linie, folosind culoarea de desen, stilul și grosimea curentă, între punctele de coordonate *(x1, y1)* și *(x2, y2)*.

void far linerel(int dx, int dy);

Desenează o linie din poziția curentă pînă la punctul aflat la distanța relativă *(dx, dy)*, deplasînd și poziția curentă în noul punct.

void far lineto(int x, int y);

Desenează o linie din poziția curentă pînă la punctul de coordonate *(x, y)*, deplasînd și poziția curentă în noul punct.

void far moverel(int dx, int dy);

Deplasează poziția curentă cu *dx* pixeli pe direcția x, respectiv cu *dy* pixeli pe direcția y.

void far moveto(int x, int y);

Deplasează poziția curentă în punctul de coordonate *(x, y)* relativ la portul de afișare curent.

void far rectangle (int st, int sus, int dr, int jos);

Desenează un dreptunghi, folosind culoarea de desen, stilul și grosimea curentă, parametrii *(st, sus)* specificînd coordonatele vîrfului stînga sus, iar *(dr, jos)* ale vîrfului dreapta jos.

void far bar(int st, int sus, int dr, int jos);

Desenează o bară dreptunghiulară bidimensională, fără contur, umplută în funcție de modul și culoarea de umplere curentă. Parametrii *(st, sus)* dau coordonatele vîrfului stînga sus, iar *(dr, jos)* ale vîrfului dreapta jos.

void far bar3d(int st, int sus, int dr, int jos, int adincime, int acoper);

Desenează o bară dreptunghiulară tridimensională, conturată, umplută în funcție de modul și culoarea de umplere curentă, adîncimea sa fiind specificată de parametrul *adincime* (dacă *adincime* = 0, se desenează o bară bidimensională conturată). Parametrii *(st, sus)* dau coordonatele vîrfului stînga sus, iar *(dr, jos)* ale vîrfului dreapta jos. Parametrul *acoper* specifică dacă bara va fi acoperită (diferit de zero) sau nu (0).

void far fillelipse(int x, int y, int razax, int razay);

Desenează o elipsă de centru *(x, y)*, de rază orizontală *razax*, respectiv verticală *razay*, umplută în funcție de modul și culoarea de umplere curentă.

void far fillpoly(int nr_virf, int far *puncte);

Desenează conturul unui poligon cu *nr_virf* virfuri, folosind culoarea de desen curentă, umplut în funcție de modul și culoarea de umplere curentă.

Parametrul ***puncte*** pointează la o secvență de (*nr_virf* * 2) întregi, fiecare pereche reprezentînd coordonatele *x* și *y* ale cîte unui vîrf.

void far floodfill(int x, int y, int margine);

Umple o zonă ecran închisă, în funcție de modul și culoarea de umplere curentă. Coordonatele *(x, y)* specifică un punct din zonă.

void far pieslice(int x, int y, int unghi1_1, int unghi_2, int rad);

Desenează conturul și umple (folosind pozițiile curente) un sector de cerc, de centru *(x, y)*, între *unghi_1* și *unghi_2*, de rază *rad*. Valoarea unghiului crește în sens invers celui orar, 0 grade fiind pe ora 3, 90 de grade pe ora 12, etc.

void far sector(int x, int y, int unghi1_1, int unghi_2, int razax, int razay);

Desenează conturul și umple (folosind pozițiile curente) un sector de elipsă, de centru *(x, y)*, între *unghi_1* și *unghi_2*, de rază orizontală *razax*, respectiv verticală *razay*. Valoarea unghiului crește în sens invers celui orar, 0 grade fiind pe ora 3, 90 de grade pe ora 12, etc.

void far setaspectratio(int xasp, int yasp);

Modifică factorul de corecție curent.

void setlinestyle(int stil, int sablon, int grosime);

Stabilește stilul de trasare al liniile pentru desenele realizate. Structure **linesettingstype** este definită în **graphics.h**, după cum urmează:

```
struct linesettingstype
{
        int linestyle;
        unsigned pattern;
        int thickness;
}
```

Parametrul *stil* (corespunzător lui **linestyle**) specifică modul de trasare al liniilor. Acești operatori sînt definiți prin enumerarea **line_style** din **graphics.h**, după cum urmează:

Nume	Valoare	Descriere
SOLID_LINE	0	Linie continuă
DOTTED_LINE	1	Linie punctată
CENTER_LINE	2	Linie centrată
DASHED_LINE	3	Linie întreruptă
USERBIT_LINE	4	Stil definit de utilizator

Parametrul *grosime* (corespunzător lui **thickness**) specifică grosimea liniilor, astfel

Nume	Valoare	Descriere
NORM_WIDTH	1	Grosime un pixel
THICK_WIDTH	3	Grosime trei pixeli

Parametrul *sablon* (corespunzător lui **pattern**) are semnificație doar dacă *stil* are valoarea **USERBIT_LINE**. În acest caz, dacă un bit din *sablon* este poziționat pe 1, pixelul corespunzător din linie este desenat utilizînd culoarea de desen curentă.

Dacă funcția primește la intrare un parametru invalid, **graphresult** returnează -1, iar stilul curent nu se modifică.

void far setfillstyle(int sablon, int culoare);

Stabilește modul și culoarea de umplere curente. Enumerarea **fill_pattern** din **graphics.h** denumește modurile de umplere.

Dacă funcția primește la intrare un parametru invalid, **graphresult** returnează -11, iar modul și culoarea curente de umplere nu se modifică.

void far setfillpattern(char far *sablon, int culoare);

Funcția este asemănătoare cu **setfillstyle**, dar se folosește pentru selectarea unui *sablon* 8*8 definit de utilizator.

void far getaspectratio(int far *xasp, int far *yasp);

Returnează în *xasp*, respectiv *yasp*, factorii de corecție pentru modul grafic curent (raportul lor fiind factorul de scară utilizat de sistemul grafic pentru ca cercurile să fie cu adevărat rotunde pe ecran) pe cele două coordonate.

void far getfillpattern(char far *sablon);

Copiază un model de umplere definit de utilizator, stabilit cu **setfillpattern**, într-o zonă de memorie de 8 biți pointată de *sablon*. Fiecare bit corespunde unui unui pixel din *sablon*. Dacă un bit este poziționat pe 1, pixelul corespunzător va fi afișat.

void far getfillsettings(struct fillsettingstype far *info);

Completează o structură de tipul **fillsettingstype** pointată de *info* cu informații despre modelul și culoarea curentă de umplere. Acest tip este definit în **graphics.h** după cum urmează:

```
struct fillsettingstype
{
        int pattern;            /* model de umplere */
        int color;              /* culoare de umplere */
}
```

A1.2.3. Tratarea ecranului

void far cleardevice(void);

Șterge întregul ecran grafic (umple cu culoarea de fond) și stabilește poziția curentă în punctul de coordonate (0, 0).

void far setactivepage(int pagina);

Stabilește pagina grafică activă, cea la care vor fi redirectate toate ieșirile grafice (aceasta poate să fie sau nu vizibilă). Doar plăcile grafice EGA, VGA și Hercules permit mai multe pagini grafice.

void far setvisualpage(int pagina);

Stabilește pagina grafică vizibilă.

void far clearviewport(void);

Șterge portul de afișare (umple cu culoarea de fond) și stabilește poziția curentă în punctul de coordonate (0, 0) relativ la portul de afișare.

void far getviewsettings(struct viewporttype far *port);

Completează o structură de tip **viewporttype** pointată de *port* cu informații despre portul curent de afișare. Structura **viewporttype** este definită în **graphics.h** după cum urmează:

```
struct viewporttype
{
        int left, top, right, bottom;      /* coordonate virfuri */
        int clip;                          /* taiere */
};
```

void far setviewport(int stinga, int sus, int dreapta, int jos, int taiere);

Stabilește noul port de afișare, avînd coordonatele colțurilor stînga sus *(stinga, sus)*, respectiv dreapta jos *(dreapta, jos)*. Poziția curentă este stabilită în punctule de coordonate relative (0, 0). Parametrul *taiere* determină dacă desenele vor fi trunchiate (0) sau nu (diferit de 0) la limitele portului stabilit. Dacă se specifică parametri invalizi, **graphresult** returnează valoarea -11, iar poziționările curente nu se modifică.

void far getimage(int stinga, int sus, int dreapta, int jos, void far *destin);

Copiază o imagine (bit map) dintr-o zonă ecran dreptunghiulară definită de *stinga, sus, dreapta, jos* într-o zonă de memorie pointată de *destin*. Primele două cuvinte din această zonă sînt urilizate pentru lățimea și înălțimea dreptunghiului, iar restul pentru imaginea însăși.

unsigned far imagesize(int stinga, int sus, int dreapta, int jos);

Returnează numărul de octeți necesari pentru a memora imaginea dintr-o zonă ecran dreptunghiulară definită de *stinga, sus, dreapta, jos*. Dacă valoarea este mai mare decît 64 Kocteți - 1, funcția returnează 0xffff (-1).

void far putimage(int stinga, int sus, void far *sursa, int oper);

Reface imaginea salvată cu **getimage** într-o zonă ecran dreptunghiulară avînd coordonatele stînga sus *(stinga, sus)*. **sursa* pointează la zona de memorie unde este stocată imaginea. Parametrul oper specifică modul de calculare al culorii elementelor de imagine, pe baza culorii elementului de pe ecran și a celui corespunzător din memorie. Acești operatori sînt definiți prin enumerarea **putimage_ops** di graphics.h, după cum urmează:

Nume	Valoare	Descriere
COPY_PUT	0	Copiere
XOR_PUT	1	Sau Exclusiv
OR_PUT	2	Sau
AND_PUT	3	Și
NOT_PUT	4	Negație

unsigned far getpixel(int x, int y);

Returnează culoarea elementului de imagine aflat în poziția de coordonate *(x, y)*.

void far putpixel(int x, int y, int culoare);

Afișează un element de imagine de culoarea *culoare* în poziția de coordonate *(x, y)*.

A1.2.4. Afișarea textului

void far gettextsettings(struct texsettingstype far *info);

Completează o structură de tipul **texsettingstype** pointată de *info* cu informații despre setul de caractere, direcția, dimensiunea și alinierea lor. Această structură este definită în **graphics.h** după cum urmează:

```
struct textsettingstype
{
        int font;
        int direction;
        int charsize;
        int horiz;
        int vert;
{;
```

void far outtext(char far *sir);

Scrie șirul specificat de *sir* în portul de afișare curentă, începînd de la poziția curentă, utilizînd poziționările curente pentru textul grafic. Dacă direcția este *HORIZ_DIR* și alinierea *LEFT_TEXT*, poziția curentă se mută spre stînga cu valoarea **textwidth(sir)**. În caz contrar, ea nu se modifică.

void far outtextxy(int x, int y, char far *sir);

Scrie șirul specificat de *sir* în portul de afișare curentă, începînd de la poziția de coordonate *(x, y)*, utilizînd poziționările curente pentru textul grafic.

int registerbgifont(void (*set)(void));

Pemite încărcarea unui set de caractere și 'înregistrarea' sa. După înregistrare, rezultatul funcției **registerbgidriver** (valoarea numărului pentru setul de caractere) trebuie tranmis la **settextstyle**. În caz de eroare, se returnează o valoare negativă.

În mod normal, rutina **settextstyle** încarcă un set de caractere alocînd memorie și încărcînd fișierul corespunzător de pe disc. O alternativă a acestei scheme dinamice este legarea unuia sau mai multor fișiere pentru seturi de caractere în fișierul executabil. Pentru aceasta, fișierele CHR vor fi convertite

în fișiere **.OBJ** (rutina **BGIOBJ**), iar în program se vor plasa apeluri ale rutinei **registerbgifont** (înainte de apelul funcției **settextstyle**) pentru a înregistra seturile de caractere. La construirea programului, în fișierul .OBJ vor trebui legate fișierele .OBJ corespunzătoare seturilor trasate.

void far settextjustify(int horiz, int vert);

Stabilește alinierea textului grafic scris, orizontal și vertical,, în jurul poziției curente. Enumerarea **text_just** din **conio.h** definește constantele simbolice pentru această operație:

Nume	Valoare	Descriere
LEFT_TEXT	0	Orizontal (implicit)
CENTER_TEXT	1	Orizontal și vertical
RIGHT_TEXT	2	Orizontal
BOTTOM_TEXT	0	Vetical (implicit)
TOP_TEXT	2	Vertical

void far settextstyle(int set, int directie, int dim);

Stabilește setul de caractere, direcția de scriere și dimensiunea lor. Parametrul *set* specifică setul de caractere. În fișierul **graphics.h** este definită enumerarea **font_names**, cu următoarele constante simbolice:

Nume	Valoare	Descriere
DEFAULT_FONT	0	Set mapat 8*8 (implicit)
TRIPLEX_FONT	1	Set trasat triplex
SMALL_FONT	2	Set trasat small
SANSSERIF_FONT	3	Set trasat sans-serif
GOTHIC_FONT	4	Set trasat gotic

Biblioteca grafică cuprinde un set de caractere mapat pe bit (8*8) și mai multe seturi trasate. În cazul seturilor mapate, fiecare caracter este definit printr-o matrice de elemente de imagine. În cazul seturilor trasate, caracterele sînt definite printr-un set de vectori care specifică sistemului grafic modul de desenare a caracterului.

Avantajul celui de-al doilea tip este evident pentru caractere mari. Setul fiind definit prin vectori, rezoluția se va păstra și pentru dimensiuni mari. În cazul seturilor mapate, matricea este multiplicată cu un factor de scală. Cu cît acesta crește, rezoluția caracterelor devine tot mai slabă.

Dacă trunchierea este validată, orice șir de caractere va fi tăiat la marginea portului de afișare. În caz contrar, șirurile formate din caractere mapate nu vor fi afișate dacă depășesc marginile ecranului, iar cele formate din caractere trasate vor fi trunchiate la marginea ecranului.

Setul mapat implicit este înglobatt în pachetul grafic, fiind disponibil în momentul execuției programului. Seturile trasate sînt păstrate în fișiere **.CHR** separate, care pot fi încărcate pentru execuție și convertite în fișiere **.OBJ** (utilitarul **BGIOBJ**), iar apoi legate în fișierul **.EXE**.

Parametrul *directie* specifică direcția de scriere, astfel:

Nume	Valoare	Descriere
HORIZ_DIR	0	Orizontal (implicit)
VERT_DIR	1	Vertical

Parametrul *dim* specifică factorul de scală. Dacă este egal cu 1, funcțiile **outtext** și **outtextxy** vor afișa caracterele din setul mapat 8*8 într-un dreptunghi de ecran de 8*8 elemente de imagine. Dacă este egal cu 2, caracterele acestui set vor fi afișate într-un dreptunghi de ecran de 16*16 elemente de imagine, ș.a.m.d. Dacă parametrul are valoarea 0, cele două funcții vor utiliza pentru seturile trasate fie factorul de scală implicit (4), fie cel specificat prin funcția **setusercharsize**.

void far setusercharsize(int multx, int divx, int multy, int divy);

Permite stabilirea factorilor de scară pentru seturi de caractere trasate. Lățimea implicită a caracterelor este scalată cu *multx:divx*, iar înălțimea implicită cu *multy:divy*. Funcția are efect doar dacă parametrul *dim* din apelul anterior al funcției **settextstyle** are valoarea 0.

int far textheight(char far *sir);

Returnează înălțimea în elemente de imagine a șirului specificat de *sir*, pe baza dimensiunii setului curent de caractere și a factorului de multiplicare.

int far textwidth(char far *sir);

Returnează lățimea în elemente de imagine a șirului specificat de *sir*, pe baza dimensiunii setului curent de caractere și a factorului de multiplicare.

A1.2.5. Controlul culorilor

Ecranul grafic constă dintr-un tabel de elemente de imagine (pixeli), fiecare generînd un unic punct colorat pe ecran. Valoarea elementului de imagine nu specifică direct culoarea, ci un index într-o tabelă a culorilor numită **paletă**. Intrarea în paletă corespunzătoare valorii unui anumit element de imagine conține informația exactă de culoare pentru acel element.

Ca urmare, deși partea de hardware ar putea afișa multe culori, doar un anumit subset poate fi practic afișat la un moment dat, numărul lor fiind egal cu numărul intrărilor în paletă. De exemplu, la EGA, partea de hardware poate afișa 64 de culori diferite, dar numai 16 la un moment dat. Dimensiunea paletei EGA este 16. Un element de imagine va putea lua o valoare cuprinsă între **0** și **dimensiune paletă-1**.

În mod curent, se utilizează termenul **culoare**. De fapt, această culoare este o valoare a unui element de imagine, respectiv un index în paletă. Doar paleta determină culoarea reală de pe ecran. Lucrînd asupra paletei, pot fi modificate culorile afișate pe ecran, chiar dacă valorile elementelor de imagine nu se modifică.

Culoarea de fond corespunde întotdeauna unei valori 0 a elementului de imagine. Ștergerea unei zone de ecran înseamnă poziționarea pe 0 a valorilor tuturor elementelor de imagine din acea zonă.

Culoarea de desen este valoarea care o iau elementele de imagine la trasarea unei linii.

int far getmaxcolor(void);

Returnează valoarea maximă validă pentru culoare, corespunzător modului și driverului grafic curent.

void far getpalette(struct palettetype far * paleta);

Completează o structură de tipul **palettetype** pointată de *paleta* cu informații referitoare la paleta curentă. Această structură este definită în **graphics.h**, astfel:

```
struct palettetype
{
        unsigned char size; /* numărul de culori din paletă, corespunzător
                               modului şi driverului grafic curent*/
        signed char colors[MAXCOLORS + 1];   /* tabel de dimensiune
                               size, conţinînd culoarea actuală pentru fiecare
                               intrare în paletă */
};
```

int far getpalettesize(void);

Returnează numărul de intrări în paleta curentă.

void far setbkcolor(int culoare);

Stabileşte culoarea de fond curentă.

void far setcolor(int culoare);

Stabileşte culoarea de desen curentă.

void far setallpalette(struct palettetype far *paleta);

Stabileşte paleta curentă la valoarea specificată de structura **palettetype** pointată de *paleta*.

void far setpalette(int nr_cul, in culoare);

Modifică intrarea *nr_cul* din paletă la valoarea dată de *culoare*.

Controlul culorilor la CGA

În acest caz, se poate lucra cu rezoluţie mică (320*200), ceea ce permite afişarea a patru culori, sau cu rezoluţie mare (640*200), utilizîndu-se doar două culori.

În modurile de rezoluţie mică, se poate alege între 4 palete predefinite, fiecare avînd 4 culori. Doar prima intrare (culoarea 0), reprezentînd culoarea de fond, poate fi modificată, restul (1, 2, 3) fiind fixe.

Această culoare de fond poate fi una din cele 16 disponibile, definite în fişierul **graphics.h** după cum arată tabelul de mai jos.

Pentru stabilirea culorii de fond, se utilizează funcţia **setbkcolor(n)**, unde *n* este una din intrările din tabelul de mai sus. De notat că în cazul CGA, această valoare nu este o valoare de element de imagine, ci culoarea reală care va fi introdusă în prima intrare din paletă.

Valoare numerică	Constantă simbolică
0	BLACK
1	BLUE
2	GREEN
3	CYAN
4	RED
5	MAGENTA
6	BROWN
7	LIGHTGRAY
8	DARKGRAY
9	LIGHTBLUE
10	LIGHTGREEN
11	LIGHTCYAN
12	LIGHTRED
13	LIGHTMAGENTA
14	YELLOW
15	WHITE

Paleta se alege prin specificarea modului (CGAC0, CGAC1, CGAC2, CGAC3), acestea utilizînd paletele 0 - 3, după cum indică tabelul următor. Culorile şi constantele simbolice corespunzătoare sînt definite în **graphics.h**.

În modurile de înaltă rezoluţie, sînt afişate două culori: fondul negru şi culoarea de desen, elementele de imagine putînd lua valorile 0 şi 1. Datorită unei anomalii constructive, culoarea de desen este considerată de partea de hardware ca fiind culoare de fond, astfel încît ea va fi stabilită cu funcţia **setbkcolor**. Culorile disponibile sînt cele din tabelul precedent, acestea fiind utilizate pentru afişarea tuturor elementelor de imagine avînd valoarea 1.

Modurile video corespunzătoare acestui caz sînt CGAHI, MCGAMED, MCGAHI, ATT400MED şi ATT400HI.

Întrucît paleta CGA este predefinită, nu este recomandată utilizarea funcției **setallpalette**.

Controlul culorilor la EGA și VGA

În acest caz, paleta conține 16 intrări din 64 de culori posibile, fiecare putînd fi modificată de utilizator.

Paleta implicită corespunde celor 16 culori CGA: negru=0,..., alb=15. În fișierul **graphics.h** sînt definite constantele simbolice corespunzătoare: **EGA_BLACK**,..., **EGA_WHITE**, valori care pot fi obținute cu funcția **getpalette**.

Rutina **setbkcolor(n)** se comportă diferit față de CGA. Astfel, ea copiază valoarea culorii din intrarea *n* în paletă în intrarea 0. În ceea ce privește culorile, driverul VGA se comportă la fel ca și EGA, dar are rezoluție mai bună (și elemente de imagine mai mici).

A1.2.6. Tratarea erorilor

char * far grapherrormsg(int cod_eroare);

Returnează un pointer la un șir care reprezintă mesajul de eroare asociat lui *cod_eroare*, valoare obținută cu funcția **graphresult**.

int far graphresult(void);

Returnează codul de eroare pentru ultima operație grafică.

În momentul apariției unei erori la apelul unei funcții a bibliotecii grafice, este poziționat un cod intern de eroare. Mesajul pentru codul respectiv poate fi obținut prin apelul **grapherrormsg(graphresult())**. Codurile de eroare sînt definite astfel în **graphics.h**:

Cod eroare	Constanta *graphics_error*	Mesajul de eroare corespunzător (șir)
0	grOk	Fără eroare
-1	grNoInitGraph	Sistem grafic neinițializat
-2	grNotDetected	Parte de hardware pentru grafică nedectectată

-3	grFileNotFound	Fișier driver negăsit
-4	grInvalidDriver	Fișier driver invalid
-5	grNoLoadMem	Memorie insuficientă pentru încăracrea driverului
-6	grNoScanMem	Memorie insuficientă
-7	grNoFlodMem	Memorie insuficientă
-8	grFontNotFound	Fișier pentru set de caractere negăsit
-9	grNoFontMem	Memorie insuficientă pentru încărcare setului de caractere
-10	grInvalidMode	Mod grafic invalid pentru driverul ales
-11	grError	Eroare grafică
-12	grIOerror	Eroare grafică de intrare/ieșire
-13	grInvalidFont	Fișier pentru set de caractere invalid
-14	grInvalidFontNum	Număr de set de caractere invalid
-15	grInvalidDeviceNum	Numărul de dispozitiv invalid
-18	grInvalidVersion	Versiune invalidă de fișier

Codul de eroare se modifică dor cînd o funcție grafică raportează o eroare. Codul de eroare este poziționat pe 0 doar cînd funcția **initgraph** este executată cu succes sau cînd este apelată funcția **graphresult**. Deci, pentru a se ști care funcție grafică a generat o anumită eroare, valoarea returnată de **graphresult** trebuie memorată într-o variabilă și apoi testată.

2.7. Interogarea stării

void far getarccoords(struct arccordstype far *str);

Completează o structură de tip **arccordstype** pointată de *str* cu informații despre ultimul apel al funcției **arc**. Structura este definită în **graphics.h** astfel:

```
struct arccorsdstype
{
        int x, y;               /* cordonatele centrului */
        int xstart, ystart;     /* coordonatele de început */
        int xend, yend;         /* coordonatele de sfirsit */
};
```

void far getaspectratio(int far *xasp, int far *yasp);

Returnează în **xasp* și **yasp* factorul de corecție al modului grafic curent, pe cele două coordonate (factorul de scară utilizat de sistemul grafic pentru ca cercurile să fie cu adevărat rotunde pe ecran). În afară de cazul monitoarelor VGA, există relația **xasp* <= **yasp*, întrucît elementele de imagine au dimensiune mai mare pe verticală decît pe orizontală (în cazul VGA, cele două valori sînt egale). În general, **yasp* este normalizat la 10.000.

int far getbkcolor(void);

Returnează culoarea de fond curentă.

int far getcolor(void);

Returnează culoarea de desen curentă.

struct palettetype * far getdefaultpalette(void);

Returnează un pointer la paleta implicită care a fost poziționată de driverul curent la inițializarea sa.

char * far getdrivername(void);

Returnează un pointer la un șir care specifică driverul curent.

void far getfillpattern(char far *model);

Copiază modelul de umplere stabilit prin funcția **setfillpattern** într-o zonă de 8 octeți pointată de *model*, fiecare octet corespunzînd la 8 elemente de imagine din model. Dacă bitul din octetul din *model* are valoarea 1, elementul de imagine corespunzător este afișat pe ecran.

void far getfillsettings(struct fillsettingstype far *info);

Completează o structură de tip **fillsettingstype** pointată de *info* cu informații despre modelul și culoarea curentă de desen (de umplere). Structura este definită în **graphics.h** după cum urmează:


```
struct fillsettingstype
{
        int pattern;            /* modelul curent de umplere */
        int color               /* culoarea curentă de umplere */
};
```

int far getgraphmode(void);

Returnează modul grafic stabilit de **initgraph** sau **setgraphmode**.

int far getmaxmode(void);

Returnează numărul maxim de mod pentru driverul curent, direct din driver, ceea ce oferă un avantaj față de funcția **getmoderange**, care lucrează doar cu drivere Borland.. Numărul minim pentru mod este 0.

int far getmaxx(void);

Returnează valoarea maximă pentru coordonata *x* (relativ la ecranul întreg) pentru driverul grafic și modul curent.

int far getmaxy(void);

Returnează valoarea maximă pentru coordonata *y* (relativ la ecranul întreg) pentru driverul grafic și modul curent.

char * far getmodename(int mod);

Returnează un pointer la un șir care conține numele modului grafic specificat de *mod*.

void far getmoderange(int driver, int far *modlo, int far *modhi);

Returnează intervalul de moduri grafice valide (minim îm *modlo*, maxim în *modhi*) pentru driverul grafic specificat de *driver*. Dacă se specifică un driver invalid, parametrii **modlo* și **modhi* vor lua valoarea -1. Dacă *driver* are valoarea -1, se returnează valorile de mod pentru driverul curent.

int far getx(void);

Returnează coordonata *x* a poziției grafice curente.

int far gety(void);

Returnează coordonata *y* a poziției grafice curente.

Bibliografie

Turbo C V 2.0	- Reference Guide vol. I. & II.
Wirth N	- Algorithms + Data Structures = Programs.
Robin J., Stewart I.	- The art of C Programming.
Kernigham B., Ritchie D	- The C Programming Language.
Stephan D.	- Turbo C: The art of Programming Design, Optimization and Debugging.
LaFore R.	- Turbo C Programming for the IBM.